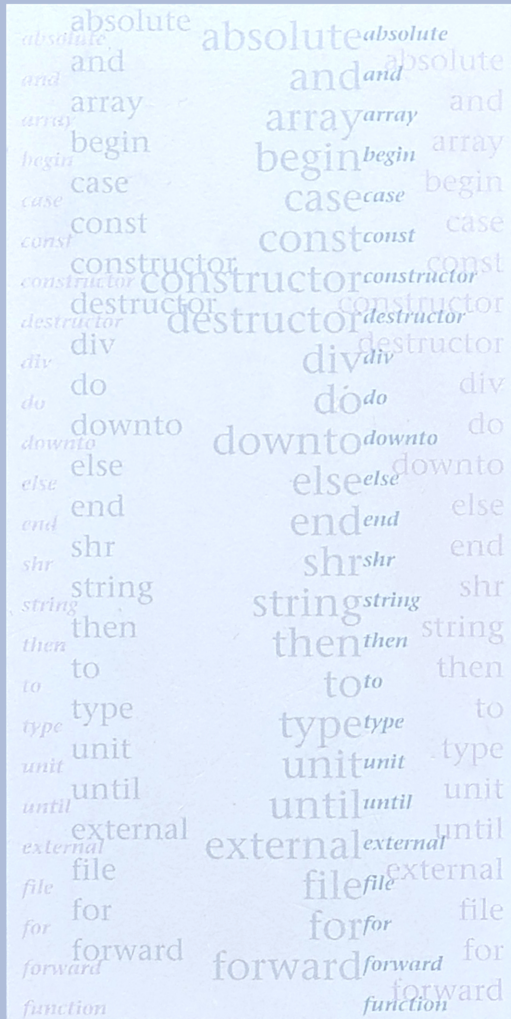


Turbo Pascal 6.0



Programmier-
handbuch

Das produktive Pascal-Entwicklungssystem für objektorientierte Programmierung. Komplett mit mausgesteuerter Benutzeroberfläche, integriertem Debugger und Assembler.

Turbo Pascal 6.0

Turbo Pascal 6.0

Programmierhandbuch

B O R L A N D

Copyright © 1990 by Borland International, INC.

Jede Vervielfältigung dieses Handbuches sowie des Turbo Pascal Softwareprogrammes wird strafrechtlich verfolgt. Die Rechte an der Dokumentation und die Rechte an dem Softwareprogramm Turbo Pascal liegen bei Borland International.

Der rechtmäßige Erwerb der Programmdisketten und der Handbücher erlaubt die Nutzung der Programme analog der Benutzung eines Buches. Entsprechend der Unmöglichkeit, daß ein Buch zugleich an verschiedenen Orten von mehreren Personen gelesen wird, darf das Softwareprogramm Turbo Pascal nicht gleichzeitig von verschiedenen Personen an verschiedenen Orten und auf verschiedenen Geräten benutzt werden. Diskettenkopien dürfen lediglich zum Zweck der Datensicherung angefertigt werden.

Sie (der Lizenznehmer von Turbo Pascal) dürfen eigene Programme, die Sie mit dem Turbo-Pascal-Sprachcompiler geschrieben und kompiliert haben, ohne Einschränkung oder Verpflichtung zu zusätzlichen Lizenzgebühren verwenden, weitergeben oder verkaufen, solange diese Programme einen Copyright-Vermerk enthalten. Unter Copyright-Vermerk verstehen wir entweder Ihren eigenen Copyright-Vermerk oder gegebenenfalls einen Hinweis wie »Created using © Borland International 1990«. Die Originaldisketten von Turbo Pascal beinhalten einige Dateien mit Hardware- und Zeichensatzinformationen, die von dem Standard-Grafik-Unit (GRAPH.TPU) verwendet werden. Diese Dateien mit den Suffixen .CHR und .BGI unterliegen besonderen Nutzungsbeschränkungen. Sie dürfen nicht als Dateien weiterverkauft werden. Sie dürfen diese Dateien aber in Ihren eigenen mit Turbo Pascal geschriebenen und kompilierten Programmen verwenden. Sie dürfen diese kompilierten Programme auch weiterverkaufen, solange sie einen Copyright-Vermerk beinhalten.

Die Beispielprogramme von Turbo Pascal, die auf den Originaldisketten mitgeliefert werden, demonstrieren die verschiedenen Eigenschaften von Turbo Pascal. Sie dienen ausschließlich zu Lehrzwecken. Sie dürfen diese Beispielprogramme editieren oder für den eigenen Gebrauch modifizieren. Sie dürfen diese Programme jedoch nicht als Ganzes oder in Teilen in kompilierter Form oder als Quelltext weitergeben oder verkaufen. Sie dürfen die Routinen aus den Beispielprogrammen in eigenen Anwendungsprogrammen verwenden, solange die resultierenden Programme nicht einem Beispielprogramm oder einem Teil eines Beispielprogrammes in Erscheinung oder Funktion gleichen.

Einschränkung der Gewährleistung
Es wird keine Garantie für die Richtigkeit des Inhaltes dieses Handbuches übernommen. Da sich Fehler, trotz aller Bemühungen, nie vollständig vermeiden lassen, sind wir für Hinweise jederzeit dankbar.

Herausgeber
Borland GmbH
Lindwurmstraße 88
Postfach 15 03 40
8000 München 2
Telefon 0 89 / 72 01 00
Telefax 0 89 / 77 93 39
Telex 5 21 26 37 mcm d

Umschlag und typografische Konzeption: Rudolf Paulus Gorbach

Grafiken: Sid Gastl, Andreas Höhne

Übertragung ins Deutsche: Dieter Strehle, Kapitel 1: Alois Stockklauser

PostScript-Belichtung: CD Computer & Dokumentations GmbH, Aalen

Druck und Bindung: Schoder Druck GmbH, Gersthofen

1. Auflage, Oktober 1990

Inhalt

Einführung 1

Was in den Kapiteln zu finden ist 1

Typographie 2

1. Sprachelemente und Konstanten 3

Spezielle Symbole und reservierte Wörter 3

Bezeichner 5

Standardbezeichner 5

Qualifizierte Bezeichner 6

Labels 7

Numerische Werte 7

Deklaration von Konstanten 10

Kommentare 11

Programmzeilen 12

2. Blöcke, Lokalität und Gültigkeitsbereich 13

Syntax 13

Gültigkeitsbereiche 15

Neudeklaration in einem untergeordneten Block 15

Der Ort einer Deklaration in einem Block 16

Neudeklaration innerhalb eines Blocks 16

Standardbezeichner 16

Der Gültigkeitsbereich von Bezeichnern im Interface-Teil 16

3. Typen 19

Einfache Typen 20

Ordinale Typen 20

Der Typ Integer 21

Der Typ Boolean 22

Der Typ Char 22

Aufzählbare Typen 22

Der Teilbereichstyp	23
Realzahlen	24
Fließkomma-Operationen über Bibliotheksroutinen	25
Fließkomma-Operationen mit einem numerischen Coprozessor	25
Der Typ String	25
Strukturierte Typen	26
Array-Typen	27
Record-Typen	28
Objekttypen	30
Mengen-Typen	35
Dateitypen	36
Zeiger-Typen	36
Prozedurtypen	37
Die Deklaration eines Prozedurtyps	38
Identische und kompatible Typen	38
Identische Typen	39
Kompatibilität von Typen	39
Zuweisungs-Kompatibilität	40
Der Typen-Definitionsteil	41

4. Variablen 43

Die Deklaration von Variablen	43
Das Daten-Segment	44
Das Stack-Segment	44
Absolute Variablen	45
Der Bezug auf Variablen	46
Die Festlegung eines Variablenbezugs	46
Arrays, Strings und Indizierungen	47
Records und Feldbezeichner	48
Bezeichner von Objektkomponenten	49
Zeiger und dynamische Variablen	49
Umwandlung von Variablentypen	49

5. Typisierte Konstanten 53

Einfache typisierte Konstanten	54
Strings	54
Strukturierte Typen	55
Arrays	55
Records	56
Konstanten mit Objekttypen	57
Mengen	57
Zeiger	58
Prozedurale Typen	58

6. Ausdrücke 61

Syntax von Ausdrücken 61

Operatoren 65

Arithmetische Operatoren 65

Logische Operatoren 66

Boolesche Operatoren 66

String-Operatoren 68

Mengen-Operatoren 68

Relationale Operatoren 69

Vergleiche einfacher Typen 69

Vergleiche von Strings 69

Vergleiche von gepackten Strings 70

Vergleiche von Zeigern 70

Vergleiche von Mengen 70

Überprüfung der Mengenzugehörigkeit 70

Der Adreß-Operator @ 70

@ mit einer Variablen 71

@ mit einem Wert-Parameter 71

@ mit einem Variablen-Parameter 72

@ mit Prozeduren und Funktionen 72

@ mit Methoden 72

Funktionsaufrufe 72

Mengenkonstruktoren 73

Umwandlung von Wert-Typen 74

Prozedur-Variablen in Ausdrücken 75

7. Anweisungen 77

Einfache Anweisungen 77

Zuweisungen 78

Prozeduraufrufe 79

Die Anweisung goto 80

Strukturierte Anweisungen 81

Verbundanweisungen 81

Bedingte Anweisungen 82

Die Anweisung if 82

Die Anweisung case 83

Schleifen 85

Die repeat-Anweisung 85

Die while-Anweisung 86

Die for-Anweisung 88

Die with-Anweisung 88

8. Prozeduren und Funktionen 91

Prozedurdeklarationen 91

Near- und far-Deklarationen 92

Deklarationen mit interrupt 93

Deklarationen mit forward 94

Deklarationen mit external 95

Deklarationen mit assembler 95

Deklarationen mit inline 96

Funktionsdeklarationen 96

Deklarationen von Methoden 98

Konstruktoren und Destruktoren 99

Parameter 101

Wertparameter 102

Variablenparameter 102

Untypisierte Variablenparameter 103

Prozedurtypen 104

Prozedur-Variablen 104

Technische Voraussetzungen 105

Prozedur-Variablen in Strukturen 106

Prozedur-Variablen als Parameter 107

9. Programme und Units 109

Die Syntax eines Programms 109

Der Programmkopf 109

Die Anweisung uses 110

Die Syntax von Units 111

Der Kopf einer Unit 112

Der Interface-Teil 112

Der Implementations-Teil 113

Der Initialisierungs-Teil 114

Units, die andere Units voraussetzen 114

uses-Anweisungen im Implementationsteil mit gegenseitigen

Abhängigkeiten 115

Gemeinsame Datendeklarationen 117

10. Ein- und Ausgabe 119

Grundlegendes 119

Standardprozeduren und Funktionen für alle Dateien 120

Standard-Prozeduren und Funktionen für Textdateien 121

Standard-Prozeduren und Funktionen für untypisierte Dateien 122

Standard-Prozeduren und Funktionen für Dateien 123

Die Variable FileMode 123

Geräte in Turbo Pascal 124
DOS-Geräteeinheiten 124
Die Geräteeinheit CON 124
Die Geräteeinheiten LPT1, LPT2 und LPT3 125
Die Geräteeinheiten COM1 und COM2 125
Die Geräteeinheit NUL 125
Text-Geräteeinheiten 125

11. Standardprozeduren und -funktionen 127

Die Prozeduren Exit, Halt und RunError 127
Prozeduren und Funktionen zur dynamischen Verwaltung des Speichers 127
Transfer-Funktionen 128
Arithmetische Funktionen 128
Ordinale Prozeduren und Funktionen 129
Stringprozeduren und -funktionen 129
Zeiger- und Adreß-Funktionen 129
Verschiedenartige Standardprozeduren und -funktionen 130

12. Standard-Units 131

Durch Standard-Units vorausgesetzte Units 132
Die Unit System 133
Die Unit Printer 136
Die Unit Dos 136
Konstanten, Typen und Variablen 136
Flag-Konstanten 136
Modi der Dateibearbeitung 137
Dateitypen und -records 137
Dateiattribut-Konstanten 138
Der Record-Typ Registers 138
Der Record-Typ DateTime 138
Der Record-Typ SearchRec 139
Die Variable DosError 139
Prozeduren und Funktionen 140
Interrupt-Prozeduren 140
Prozeduren für Datum und Uhrzeit 140
Statusfunktionen für Disketten- und Festplattenlaufwerke 140
Prozeduren zur Bearbeitung von Dateieinträgen 140
Prozeduren und Funktionen für Prozesse 140
Funktionen zur Ermittlung von Environment-Einträgen 141
Funktionen und Prozeduren für Parameter von DOS 141
Die Unit Crt 141
Die Dateien Input und Output 141

Fenster	142
Spezialzeichen	142
Zeilenweise Eingabe	143
Konstanten, Typen und Variablen	143
Videomodi	143
Text- und Hintergrundfarben	144
Die Variable CheckBreak	144
Die Variable CheckEof	145
Die Variable CheckSnow	145
Die Variable DirectVideo	145
Die Variable LastMode	145
Die Variable TextAttr	146
Die Variablen WindMin und WindMax	146
Prozeduren und Funktionen	147
Die Unit Graph	148
Grafiktreiber und Initialisierungen	148
Einige technische Details	149
Der Grafik-Cursor	150
Das Koordinatensystem	150
Textausgabe	151
Objekte und Zeichenstil	151
Grafikfenster, Bildausschnitte und Grafikseiten	152
Farben und Farb-Paletten	152
Speicherorganisation und Farbkontrolle für CGA-Karten	153
CGA-Karten im hochauflösenden Modus	154
Manipulation von Farb-Paletten	154
Speicherorganisation und Farbkontrolle für EGA- und VGA-Karten	154
Fehlerbehandlung	155
Einige einfache Beispiele	156
Dynamische Belegung des Speichers	158
Konstanten, Typen und Variablen	160
Prozeduren und Funktionen	165
Die Unit Overlay	168
Konstanten, Typen und Variablen	168
Prozeduren und Funktionen	169
Die Unit Turbo3	170
Interface-Teil	170
Die Variable Kbd	171
Die Variable CBreak	172
Prozeduren und Funktionen	172
Die Unit Graph3	172
Prozeduren und Funktionen	173
Turtlegrafik	173

13. Der Coprozessor 8087 175

- Die Datentypen des 8087 176
- Arithmetische Operationen im Format Extended 177
- Vergleiche von Realzahlen 178
- Der Rechenstack des 8087 179
- Die Ausgabe von Realzahlen 180
- Units und der 8087 180
- Maschinenprogramme und der Emulator 181
- Die Prüfung des 8087 beim Start des Programms 181
- Die Systemvariable Test8087 182
- 8087-Emulation mit dem Assembler 183

14. Interne Details 185

- Speicherbelegung durch ein Turbo-Pascal-Programm** 185
- Die dynamische Verwaltung des Speichers** 187
- Freigabe von dynamischem Speicherplatz 188
- Die Fragmentliste 191
- Die Variable HeapError 193
- Interne Datenformate** 194
- Integer-Typen 194
- Char-Typen 194
- Boolesche Typen 195
- Aufzählungstypen 195
- Real-Typen 195
- Der Datentyp Real 195
- Der Datentyp Single 196
- Der Datentyp Double 196
- Der Datentyp Extended 197
- Der Datentyp Comp 197
- Zeiger-Typen 198
- String-Typen 198
- Mengen 198
- Array-Typen 198
- Record-Typen 199
- Datei-Typen 199
- Prozedurale Typen 200
- Direkte Zugriffe auf Speicher-Adressen** 200
- Die Arrays Mem, MemW und MemL 200
- Das interne Datenformat von Objekten** 201
- Virtuelle Methodentabellen 202
- Die Standardfunktion SizeOf 204
- Die Standardfunktion TypeOf 204
- Der Aufruf von virtuellen Methoden 204

Aufrufkonventionen von Methoden	205
Konstrukturen und Destrukturen	206
Erweiterungen von New und Dispose	206
Methoden in Assembler	208
Fehlerbehandlung in Konstrukturen	211
Der Aufruf von Funktionen und Prozeduren	215
Variablen-Parameter	216
Wert-Parameter	216
Funktionsergebnisse	217
Aufrufe als near und far	218
Eingeschachtelte Routinen	218
Start- und Endcode von Prozeduren und Funktionen	219
Verfügbare Prozessor-Register	220
Exit-Prozeduren	220
Bearbeitung von Interrupts	222
Interrupt-Prozeduren	222
Textdatei-Gerätetreiber	224
Die Open-Funktion	226
Die InOut-Funktion	226
Die Flush-Funktion	226
Die Close-Funktion	227
Direkte Zugriffe auf I/O-Adressen	227
Automatische Optimierung des Programmcodes	228
Vorausberechnung von Konstanten	228
Gemeinsame Speicherung mehrfach verwendeter Konstanten	228
Auswertung zusammengesetzter boolescher Ausdrücke	229
Reihenfolge der Berechnung	229
Überprüfung von Wertebereichen	230
Schiebebefehle anstelle von Multiplikationen	230
Wortweise Ausrichtung von Daten	231
Entfernung toten Codes	231
Intelligentes Linken	231
Optimierungen durch den Programmierer	233
Inc und Dec	233
Boolesche Ausdrücke anstelle von Mengen	233
Compiler-Schalter	233

15. Der integrierte Assembler und Einbinden von Assembler-Code 235

Der integrierte Assembler	235
Die Anweisung asm	235
Einsatz der Register	236
Die Syntax einer Assembler-Anweisung	237
Labels	237

Präfix-Opcodes 238
Befehlssätze 239
Assembler-Anweisungen 240
Operanden 242
Ausdrücke 243
Unterschiede zwischen Pascal- und Assembler-Ausdrücken 243
Ausdruck-Elemente 244
Ausdruck-Klassen 249
Ausdruck-Typen 251
Ausdruck-Operatoren 252
assembler-Prozeduren und Funktionen 255
Einbinden von Assembler-Code 257
Maschinenprogramme mit dem Turbo Assembler 258
Einige Beispiele 259
Ein weiterer Ausblick 263
Maschinencode mit inline 264
inline-Anweisungen 264
inline-Deklarationen 266

Index 267

Einführung

Dieses Handbuch ist für erfahrene Programmierer zum Nachschlagen gedacht. Wenn Sie bereits einer Programmiersprache mächtig sind, am besten natürlich Pascal, ist dieses Buch für Sie genau das richtige. Es enthält die Referenz zum Turbo Pascal Standard, zusammenfassende Informationen zur Laufzeitbibliothek sowie Details zu den internen Abläufen eines Turbo Pascal Programms und eine Dokumentation zum Einbinden von Assembler-Code in Turbo Pascal Programme. Wenn Sie

- noch nie programmiert haben, gleich mit welcher Programmiersprache,
- programmieren können, aber gerne Pascal lernen möchten,
- in Pascal programmieren können, aber die integrierte Entwicklungsumgebung kennenlernen möchten,

dann lesen Sie bitte nicht diese Dokumentation, sondern das *Benutzerhandbuch*.

Was in den Kapiteln zu finden ist

Das *Programmierhandbuch* ist in vier Teile gegliedert.

- Der erste Teil beschreibt den Turbo Pascal Standard und erstreckt sich über die Kapitel 1 bis 9 »Sprachelemente«, »Blöcke«, »Typen«, »Variablen«, »Typisierte Konstanten«, »Ausdrücke«, »Anweisungen«, »Prozeduren und Funktionen«, »Programme und Units«.

- Der zweite Teil enthält eine Zusammenfassung der Laufzeitbibliothek:

Kapitel 10 »Ein- und Ausgabe« beschreibt die Prozeduren und Funktionen zur Ein- und Ausgabe, die in Turbo Pascal eingebaut sind.

Kapitel 11 »Standard-Prozeduren und -Funktionen« gibt einen Abriß der Routinen der Unit *System*, d.h. der Implementation des Standard-Sprachumfangs.

Kapitel 12 »Standard-Units« bespricht globale Variablen, Datentypen, Prozeduren und Funktionen der Standard-Units *Crt*, *Dos*, *Graph*, *Overlay* und *Printer*

sowie die beiden Units *Turbo3* und *Graph3*, die die Übernahme existierender Turbo Pascal 3.0-Programme stark vereinfachen.

Kapitel 13 »Der Coprozessor 8087« erläutert die Arbeit mit einem numerischen Coprozessor. Hier finden Sie detaillierte Informationen über die Datentypen *Single*, *Double*, *Extended* und *Comp*, die Turbo Pascal im Modus $\{N+\}$ zur Verfügung stellt - auch wenn der Computer nicht mit einem 8087, 80287 oder 80387 ausgerüstet ist.

■ Der dritte Teil beschäftigt sich mit der Belegung des Speichers durch ein Programm, den intern verwendeten Datenformaten, der dynamischen Speicher-verwaltung... kurz gesagt mit allem, was einen »typischen« Turbo Pascal-Programmierer interessiert. All das findet sich im Kapitel 14 »Interne Details«.

■ Der vierte Teil deckt alles ab, was mit Assembler-Code zusammenhängt: In Kapitel 15 »Der integrierte Assembler und Einbinden von Assembler-Code« finden sich detaillierte Informationen zum integrierten Assembler, zum Einbinden von Assembler-Code und zu inline-Code.

Typographie

Zur Verdeutlichung von Zusammenhängen verwenden wir mehrere unterschiedliche Schriftarten:

Schreibmaschine steht für Syntaxdefinitionen, Eingaben des Benutzers, Ausgaben von Beispielprogrammen und wird für Programmlistings verwendet.

Schreibmaschine **fett** kennzeichnet in Listings reservierte Wörter von Turbo Pascal, zum Beispiel **procedure**, **begin**, **object** usw.

Fettschrift wird im laufenden Text benutzt, um reservierte Wörter (wie **begin**, **end**, **object** usw.) hervorzuheben.

Kursivschrift kennzeichnen Namen von Variablen und Bezeichnern im laufenden Text. Außerdem dient diese Schriftart zur Hervorhebung neuer Begriffe.

Tasten für die Tasten Ihrer Tastatur, zum Beispiel in »Ein Druck auf Esc entfernt das Menü wieder vom Bildschirm«.

1. Sprachelemente und Konstanten

Sprachelemente (Tokens) sind die kleinsten bedeutungstragenden Einheiten eines Programms. Sie werden in die Kategorien spezielle Symbole, Bezeichner, Labels, Zahlen und Zeichenketten unterteilt.

Ein Pascal-Programm besteht aus Sprachelementen und Begrenzern, die die einzelnen Elemente voneinander trennen.

■ Ein Begrenzer kann entweder ein Leerzeichen, ein Kommentar oder ein Zeilenvorschub sein.

■ Zwei aufeinanderfolgende Sprachelemente müssen durch mindestens einen Begrenzer voneinander getrennt sein, wenn es sich bei beiden Sprachelementen um reservierte Wörter, Bezeichner, Labels oder Zahlen handelt.

■ Ein Begrenzer kann nicht Teil eines Sprachelements sein. Ausnahme: Das Sprachelement »Zeichenkette« kann Zeichen enthalten, die ansonsten als Begrenzer verwendet werden.

Spezielle Symbole und reservierte Wörter

Turbo Pascal benutzt die folgenden Teilbereiche des ASCII-Zeichensatzes:

Buchstaben – A bis Z und a bis z. Die deutschen Umlaute und das »ß« sind nicht zulässig;

Ziffern – die arabischen Ziffern von 0 bis 9;

Hexadezimalziffern – die arabischen Zahlen von 1 bis 9 sowie die Buchstaben A bis F und a bis f;

Leerzeichen – das Leerzeichen selber (ASCII-Wert 32) und alle Steuerzeichen (ASCII 0 - 31) einschließlich des Zeilenendezeichens (ASCII 13).

Spezielle Symbole und reservierte Wörter haben eine festgelegte Bedeutung (in einigen Fällen auch mehrere, abhängig vom Kontext). Die folgenden Zeichen werden von Turbo Pascal als spezielle Symbole interpretiert:

+ - * / = < > [] . , () : ; ^ @ { } \$ #

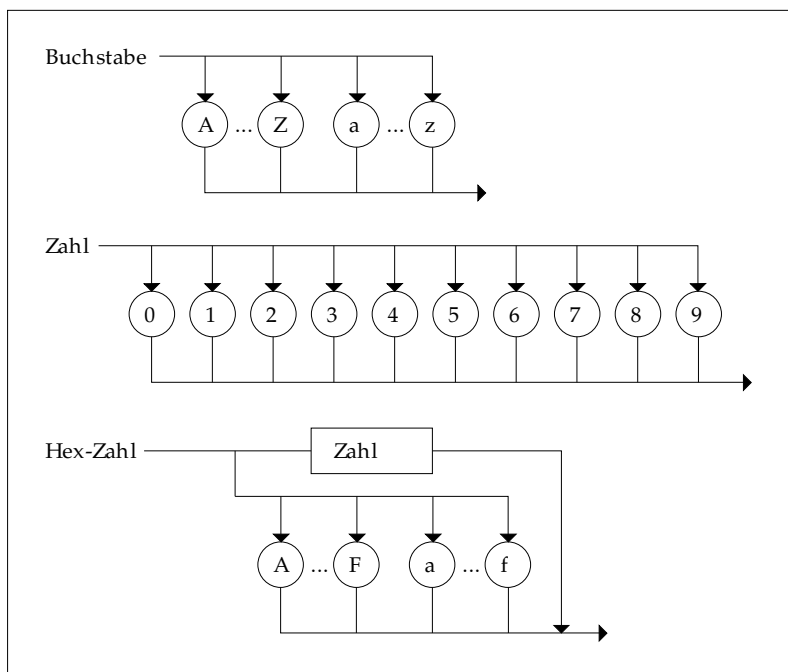


Abbildung 1.1 Von Turbo Pascal verwendete ASCII-Zeichen

Zusätzlich werden auch noch einige Paare von Symbolsymbolen als Operatoren und/oder Begrenzer verwendet:

<code>:=</code>	Zuweisungsoperator
<code><> <= >=</code>	relationale Operatoren
<code>..</code>	Teilbereiche
<code>(. .)</code>	kann als Ersatz für <code>[]</code> verwendet werden
<code>(* *)</code>	kann anstelle von <code>{ }</code> verwendet werden

Eine Unterscheidung zwischen Groß- und Kleinschreibung findet nicht statt – reservierte Wörter und alle anderen Bezeichner können also beliebig geschrieben werden. Turbo Pascal reserviert die folgenden Wörter, die in diesem Handbuch alle durch **Fettdruck** hervorgehoben sind:

and	array	asm	begin	case
const	constructor	destructor	div	do
downto	else	end	file	for
function	goto	if	implementation	
in	inline	interface	label	mod
nil	not	object	of	or
packed	procedure	program	record	repeat
set	shl	shr	string	then
to	type	unit	until	uses
var	while	with	xor	

Zusätzlich definiert Turbo Pascal die Standard-Direktiven

absolute	assembler	external	far	forward
interrupt	near	private	virtual	

Anders als reservierte Wörter können Standard-Direktiven vom Benutzer neu definiert werden. Dies sollte man jedoch vermeiden. **private** ist innerhalb einer Objekttyp-Deklaration ein reserviertes Wort.

Bezeichner

Alle Typen, Variablen, Prozeduren, Funktionen, Units, Programme und Felder von Records werden durch Bezeichner identifiziert. Ein Bezeichner kann beliebig lang sein, allerdings sind nur die ersten 63 Zeichen signifikant.

Ein Hinweis zu den Syntaxdiagrammen: Die Namen innerhalb der Kästen und Kreise stehen für die Elemente, aus denen sich ein Konstrukt zusammensetzt und sind durch Pfeile miteinander verbunden. Um ein gültiges Konstrukt zu erhalten, beginnen Sie links im Diagramm, folgen den Pfeilen (die teilweise mehrere Möglichkeiten und/oder Wiederholungen zulassen) und »addieren« die einzelnen Elemente – solange, bis Sie die rechte Seite des Diagramms erreicht haben.

Ein Bezeichner muß mit einem Buchstaben oder einem Unterstrich (ASCII-Code \$5F) beginnen und darf keine Leerzeichen enthalten. Auf das erste Zeichen können Ziffern, Buchstaben und Unterstriche folgen. Bei Bezeichnern macht Turbo Pascal ebenfalls keinen Unterschied zwischen Groß- und Kleinschreibung, die deutschen Umlaute und das »ß« sind nicht zulässig.

In diesem Handbuch sind alle Bezeichner *kursiv* gesetzt, sofern sie im Text erwähnt werden.

Standard-bezeichner

Turbo Pascal definiert bereits eine Reihe von Bezeichnern, für die deshalb der Name *Standardbezeichner* verwendet wird. Beispiele dafür sind:

```
ClearEol
GoToXY
Exit
GraphDefaults
Test8087      <- eine globale Variable
WriteLn
```

Jeder Bezeichner muß innerhalb eines Programmblocks eindeutig sein, d.h. er darf nur für eine einzige Variable, Konstante, Prozedur oder Funktion stehen.

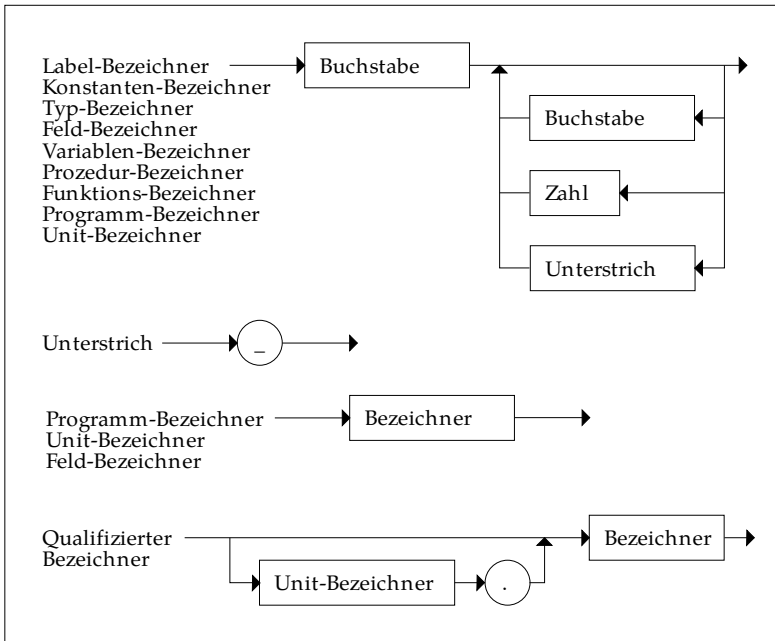


Abbildung 1.2 Syntax von Standardbezeichnern

Qualifizierte Bezeichner

Der Bezug auf Bezeichner gleichen Namens, die innerhalb verschiedener Units (siehe Kapitel 9 »Programme und Units«) deklariert sind, geschieht über eine *Qualifizierung* durch Voranstellen des Unitnamens. Wenn wir eine Variable *CursorX* annehmen, die sowohl in der Unit *Crt* als auch in der Unit *Graph* definiert ist, muß also über den Unitnamen unterschieden werden:

```
Crt.CursorX := 4; { CursorX in Unit Crt }
Graph.CursorX := 12; { CursorX in Unit Graph }
```

Analoges gilt für lokale Bezeichner einzelner Routinen: wenn die Routinen *Sub1* und *Sub2* jeweils eine lokale Variable namens *x* deklarieren, dann lassen sich diese beiden Bezeichner durch Voranstellen des Routinennamens voneinander trennen:

```
Sub1.x      { Variable x in der Routine Sub1 }
Sub2.x      { Variable x in der Routine Sub2 }
```

Hinweis: Da lokale Variablen außerhalb ihres jeweiligen Blocks physikalisch nicht existieren, läßt sich diese Erweiterung des Prinzips qualifizierter Bezeichner nur im Zusammenhang mit dem integrierten Debugger anwenden.

Für die Arbeit mit dem Debugger sind die beiden Formen qualifizierter Bezeichner auch miteinander kombinierbar:

Crt.CursorX.x { Variable x der Routine CursorX in Unit Crt }

Graph.CursorX.x { Variable x der Routine CursorX in Unit Graph }

Labels

Ein Label ist eine Zahlenfolge im Wertebereich von 0 bis 9999; führende Nullen sind nicht signifikant. Labels werden ausschließlich in Zusammenhang mit der Anweisung **goto** verwendet.

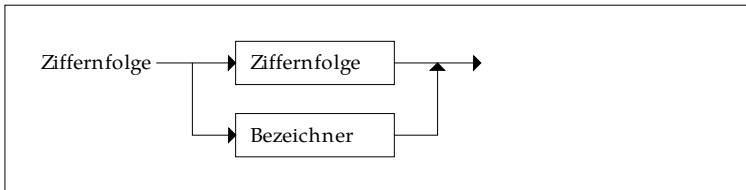


Abbildung 1.3 Syntax eines Labels

Als Erweiterung von Standard-Pascal erlaubt Turbo Pascal die Benutzung von Bezeichnern als Labels.

Numerische Werte

Wertangaben können durch das Syntaxdiagramm 1.4 formal dargestellt werden.

Für die Angabe konstanter Integer- und Realwerte wird die normale dezimale Schreibweise verwendet. Hexadezimalen Integer-Konstanten wird zur Kennzeichnung ein Dollarzeichen (\$) vorangestellt. Die wissenschaftliche Notation (E oder e, gefolgt von einem Exponenten) wird als »mal 10 hoch n« gelesen. Einige Beispiele: 7E-2 steht für $7 * 10^{-2}$; 12.25e6 und 12.25e+6 sind äquivalent und stehen für $12.25 * 10^{+6}$.

Zahlen mit Nachkommastellen oder Exponenten werden grundsätzlich als Realwerte behandelt, ganzzahlige Dezimalwerte werden als Integer gespeichert und müssen im Bereich von -2147483648 bis 2147483647 liegen. Hexadezimalzahlen sind ihrer Natur nach vorzeichenlos, d.h. das Vorzeichen ergibt sich aus der Größe des angegebenen Wertes, der zwischen \$00000000 und \$FFFFFFF liegen muß.

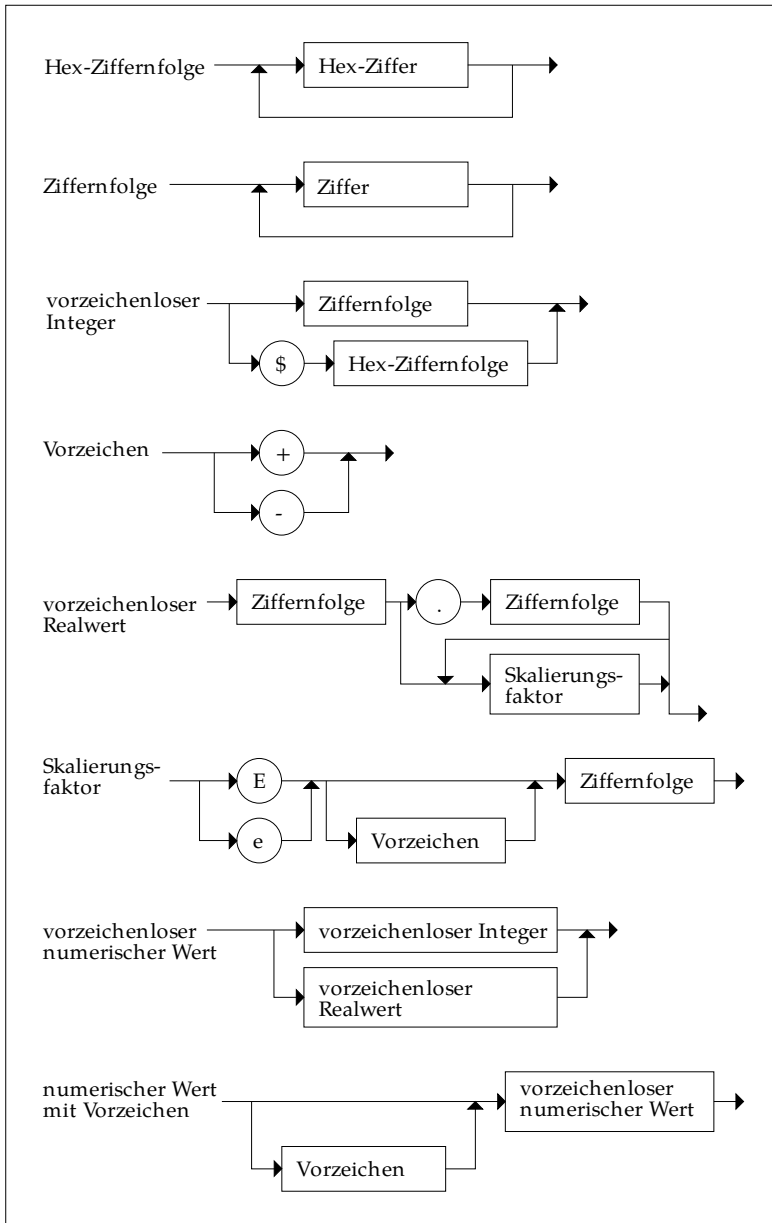


Abbildung 1.4 Syntax numerischer Werte

Zeichenketten (Strings)

Eine Stringkonstante ist eine Folge von Zeichen des IBM-Zeichensatzes (siehe Kapitel 5 »Zeichensatz- und Tastaturcodes« des *Referenzhandbuchs*), die durch Apostrophe begrenzt ist und innerhalb des Quelltextes in einer Zeile stehen muß.

Ein String kann auch leer sein, d.h. zwischen den beiden Begrenzern müssen nicht unbedingt weitere Zeichen stehen. Ein Apostroph *innerhalb* eines Strings läßt sich durch zwei aufeinanderfolgende Apostrophe darstellen.

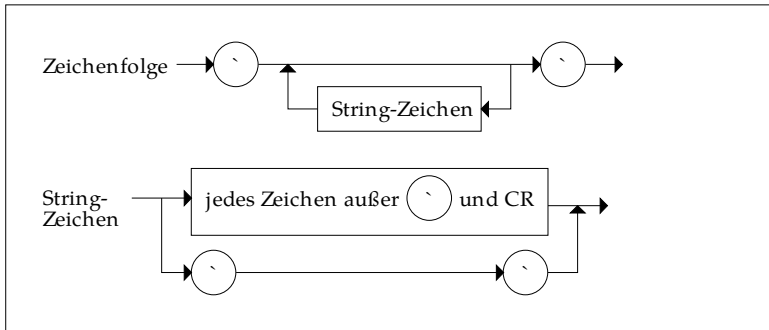


Abbildung 1.5 Syntax eines Strings

Als Erweiterung des Standards erlaubt Turbo Pascal die Aufnahme von Steuer- und Grafikzeichen in Strings: das Symbol #, gefolgt von einer vorzeichenlosen Integerkonstanten im Bereich von 0 bis 255, wird als ein Zeichen des entsprechenden ASCII-Codes interpretiert.

Zwischen dem # und der Integerkonstanten dürfen keine weiteren Zeichen stehen; mehrere aufeinanderfolgende Steuerzeichen (wie z.B. #13#10 für CR/LF) werden nicht durch zusätzliche Begrenzer voneinander getrennt.

Die Länge eines Strings ergibt sich aus der Anzahl der tatsächlich enthaltenen Zeichen: ein String mit dem Inhalt #65 hat also dieselbe Länge wie 'A', nämlich 1. Ein String von der Länge Null (eine leere Zeichenkette) ist nur zu String-Typen, ein String mit der Länge von einem Zeichen dagegen zu String- und Char-Typen kompatibel. Ein String mit mehr als einem Zeichen ist zu jedem String-Typ und jedem gepackten String-Typ kompatibel. Die folgenden Beispiele sind gültige Angaben von Strings:

```
'Turbo'   'Wie geht's'   ''''   ';'   ''
#13#10    'Zeile1'#13'Zeile2'   #7#7'Aufwachen!'#7#7
```

Deklaration von Konstanten

Eine Konstanten-Deklaration vereinbart einen Bezeichner, der innerhalb des entsprechenden Blocks für einen konstanten Wert steht. Die Wertangabe für eine Konstante kann nicht rückbezüglich sein, d.h. die Konstante selbst darf innerhalb der Wertangabe nicht erscheinen.

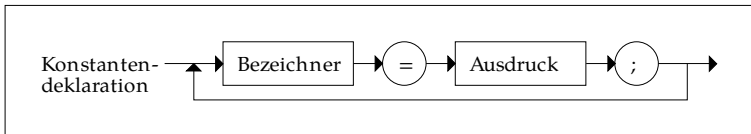


Abbildung 1.6 Syntax einer Konstanten-Deklaration

Vorzeichenbehaftete Konstanten können nur für einen Integer- oder Realwert stehen.

Während der Compiler im Standard-Pascal keine Berechnung von Konstanten vornimmt und sich **const**-Deklarationen folglich auf direkte Angaben wie

const

BUFMAX = 100;

var

Buf: **array**[0..99] **of** Integer; { [0..BUFMAX-1] ist nicht erlaubt! }

beschränken müssen, erlaubt Turbo Pascal Konstanten-Ausdrücke an jeder Stelle, an denen der Sprachdefinition nach nur eine einfache Konstante möglich ist. Damit lassen sich nicht nur Deklarationen wesentlich übersichtlicher schreiben:

const

BUFMAX = 100;

var

Buf: **array**[0..BUFMAX-1] **of** Integer; { entspricht [0..99] }

Einige weitere Beispiele dazu:

const

Min = 0; Max = 100;

Center = (Max - Min) **div** 2;

Beta = chr(255);

NumChars = Ord('Z') - Ord('A') + 1;

Message = 'Kein Speicherplatz!';

ErrMsg = 'Fehler: ' + Message;

ErrPos = 80 - Length(ErrMsg) **div** 2;

Ln10 = 2.3025890;

Ln10R = 1 / Ln10;

Numset = ['0'..'9']; Charset = ['A'..'Z'];

Alphaset = Numset + Charset;

Da der Compiler den Wert eines Konstanten-Ausdrucks während der Übersetzung bestimmen muß, gibt es natürlich auch einige Einschränkungen, die sich größtenteils selbst erklären sollten. Die folgenden Elemente dürfen in einem derartigen Ausdruck *nicht* enthalten sein:

- Werte von Variablen und typisierten Konstanten (außer von konstanten Adreß-Ausdrücken, vgl. Kapitel 5 »Typisierte Konstanten«);
- der Adreß-Operator @ (außer bei konstanten Adreß-Ausdrücken, vgl. Kapitel 5 »Typisierte Konstanten«);
- Aufrufe von Funktionen. Wie zuvor teilweise demonstriert, gibt es hier allerdings eine ganze Reihe von Ausnahmen:

Abs	Chr	Hi	Length	Lo	Odd	Ord
Pred	Ptr	Round	SizeOf	Succ	Swap	Trunc

Abgesehen von diesen Einschränkungen folgen Konstanten-Ausdrücke denselben Regeln wie in Kapitel 6 »Ausdrücke« beschrieben. Sie sind folglich auch nicht auf reine **const**-Deklarationen beschränkt:

```
const
  BufSize = 512;

type
  CharVal = Ord('A')..'Z';    { eine Untermenge }
  RecPtr  = ^DataRec;
  DataRec = record
    Prev, Next: RecPtr;
    Data: string[15];
  end;
  RecBuf  = array[0..SizeOf(DataRec)-1] of Byte;
```

Kommentare

Die beiden folgenden Konstrukte werden vom Compiler als Kommentar behandelt und bei der Compilierung ignoriert:

```
{dieser Satz steht in geschweiften Klammern}
(* das ist eine Klammer mit einem Stern *)
```

Kommentare, die als erstes Zeichen nach { bzw. (* ein Dollarzeichen (\$) enthalten, werden als Compiler-Befehle interpretiert (vgl. Kapitel 2 »Compiler-Befehle« des *Referenzhandbuches*). Beispiele:

```
{ $N+ }    { setzt den Modus N+ (IEEE-Typen) }
(* $N+ *)  { hat denselben Effekt }
```

Programmzeilen

In Turbo Pascal darf die Länge einer Programmzeile 126 Zeichen nicht überschreiten.

2. Blöcke, Lokalität und Gültigkeitsbereich

Ein Block besteht aus beliebig angeordneten Deklarationen und Anweisungen. Jeder Block ist Teil einer Prozedur-Deklaration, einer Funktions-Deklaration, eines Programms oder einer Unit. Alle Bezeichner und Labels im Deklarationsteil des Blocks sind in ihrer Wirkung auf diesen Block beschränkt – sie sind *lokal* zu diesem Block.

Syntax

Der syntaktische Aufbau jedes Blocks läßt sich durch folgendes Diagramm darstellen:

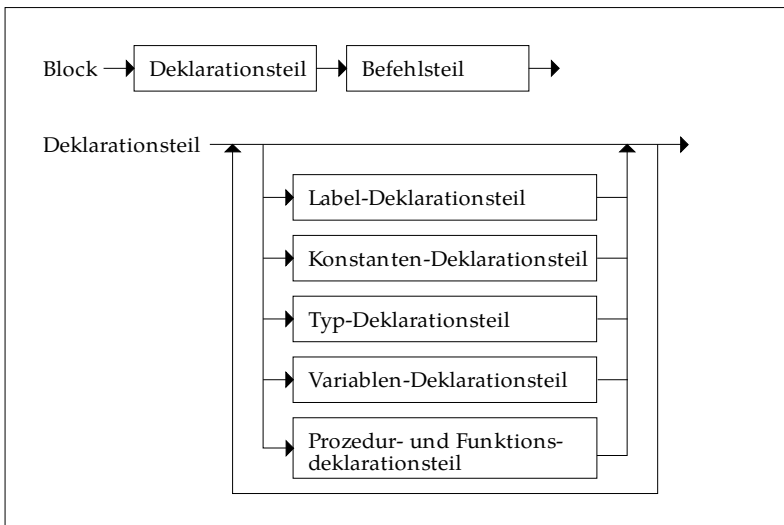


Abbildung 2.1 Syntaktischer Aufbau von Blöcken

Im *Label-Deklarationsteil* werden alle Labels vereinbart, die **goto**-Sprungziele im Anweisungsteil des betreffenden Blocks darstellen sollen. Jedes Label darf

innerhalb des Anweisungsteils nur einmal definiert werden (d.h. jedes `goto` muß ein eindeutiges Ziel haben). Ziffernfolgen, die für Labels verwendet werden, müssen im Bereich von 0 bis 9999 liegen.

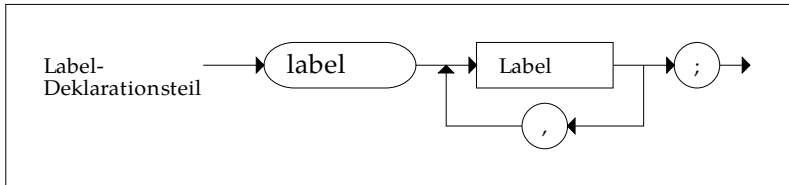


Abbildung 2.2 Syntax des Deklarationsteils eines Labels

Der *Deklarationsteil für Konstanten* beinhaltet alle Vereinbarungen von Konstanten, die lokal zum entsprechenden Block sind.

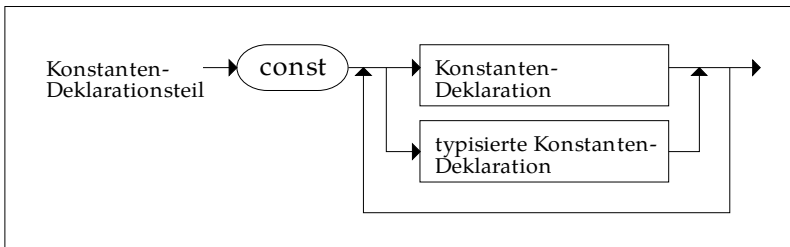


Abbildung 2.3 Syntax des Deklarationsteils für Konstanten

Der *Deklarationsteil für Typen* beinhaltet alle lokalen Typendefinitionen des dazugehörigen Blocks.

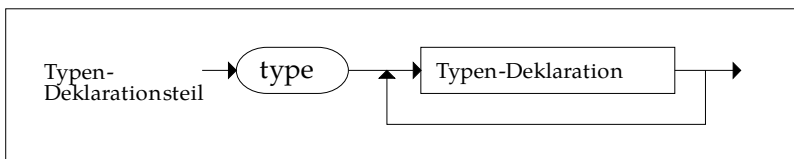


Abbildung 2.4 Syntax des Deklarationsteils für Typen

Der *Deklarationsteil für Variablen* beinhaltet alle Variablen-Deklarationen, die lokal zum entsprechenden Block sind.

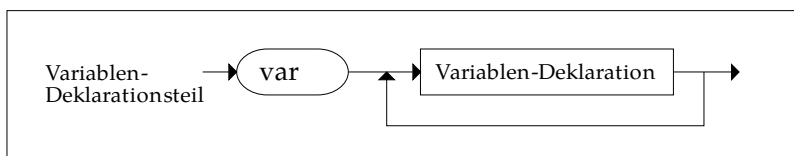


Abbildung 2.5 Syntax des Deklarationsteils für Variablen

Der *Deklarationsteil für Prozeduren und Funktionen* beinhaltet alle Prozedur- und Funktionsdeklarationen, die lokal zum entsprechenden Block sind.

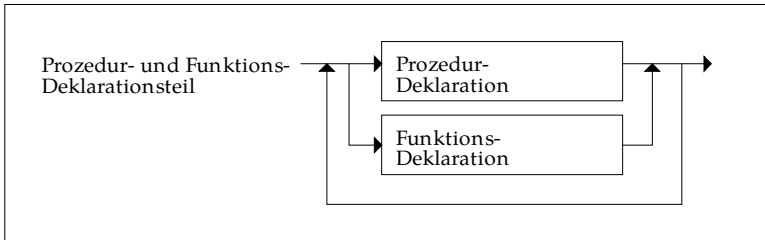


Abbildung 2.6 Syntax des Deklarationsteils für Prozeduren und Funktionen

Im *Anweisungsteil* werden schließlich alle Operationen definiert, die bei der Aktivierung des Blocks ausgeführt werden sollen.

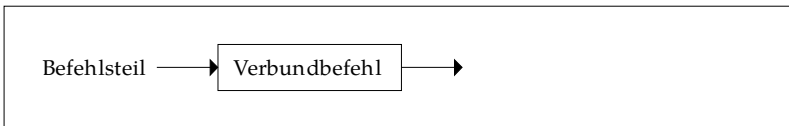


Abbildung 2.7 Syntax des Anweisungsteils

Gültigkeitsbereiche

Jeder Bezeichner und jedes Label in einer Deklaration vereinbart genau ein Objekt bzw. ein Sprungziel. Deshalb muß ein Bezeichner, genauso wie ein Label, immer im Gültigkeitsbereich (*Scope*) seiner Deklaration sein, wenn er im Programm erscheint. Der Gültigkeitsbereich von Bezeichnern und Labels liegt zwischen der eigentlichen Deklaration und dem Ende des dazugehörigen Blocks, wobei alle Blöcke mit eingeschlossen sind, die dieser Block umfaßt. Allerdings gibt es hierzu einige Ausnahmen, die in den folgenden Absätzen besprochen werden.

Neudeklaration in einem untergeordneten Block

Wenn wir davon ausgehen, daß der Block »Außen« den Block »Innen« umfaßt, d.h. ihm übergeordnet ist, dann beschränkt jede Neudeklaration eines Bezeichners von »Außen« im Block »Innen« den Gültigkeitsbereich dieses Bezeichners auf den Block »Innen« und eventuell weitere von »Innen« eingeschlossenen Blöcke. Anders gesagt: Wenn »Außen« eine Variable x deklariert

und »Innen« eine Variable gleichen Namens, dann können Anweisungen im Block »Innen« nicht auf die in »Außen« deklarierte Variable x zugreifen.

Der Ort einer Deklaration in einem Block

Bezeichner und Labels müssen deklariert sein, bevor sie in einem Block benutzt werden können. Auf Zugriffsversuche vor ihrer eigentlichen Deklaration reagiert der Compiler mit der Meldung *Unbekannter Bezeichner*. Zu dieser Regel gibt es allerdings eine Ausnahme: Der Grundtyp eines Zeigers kann tatsächlich ein noch nicht deklarierter Bezeichner sein. Aber auch dieser Bezeichner muß schließlich in derselben Typ-Deklaration erscheinen, in der auch der Zeiger steht.

Neudeklaration innerhalb eines Blocks

Bezeichner und Labels können auf der obersten Ebene eines Blocks nur jeweils einmal deklariert werden, es sei denn, ihre Neudeklaration erfolgt innerhalb eines untergeordneten Blocks oder innerhalb der Feldliste eines Records.

Der Bezeichner eines Feldes in einem Record wird innerhalb eines Record-Typs deklariert. Er ist nur dann signifikant, wenn er in Verbindung mit einem Verweis auf eine Variable desselben Record-Typs aufgeführt wird. Man kann also den Bezeichner eines Feldes innerhalb des gleichen Blocks (mit exakt demselben Namen) neu deklarieren, solange er nicht auf derselben Ebene innerhalb desselben Record-Typs steht. Ein bereits deklarierter Bezeichner kann im selben Block als Feldbezeichner erneut deklariert werden.

Der Gültigkeitsbereich einer Objekttyp-Komponente (Feld, Methode, Konstruktor, Destruktor) geht über deren Deklarationsort hinaus. Detaillierte Informationen dazu finden Sie in Kapitel 3 »Typen«.

Standardbezeichner

Turbo Pascal bietet eine ganze Reihe vordefinierter Konstanten, Typen, Prozeduren und Funktionen, die so arbeiten, als wären sie innerhalb eines Blocks deklariert worden, der das ganze Programm umschließt. Folglich umfaßt ihr Gültigkeitsbereich das gesamte Programm.

Der Gültigkeitsbereich von Bezeichnern im Interface-Teil

Sämtliche »öffentlichen« Bezeichner einer Unit, die durch die Anweisung **uses** in ein Programm oder den Interface-Teil einer anderen Unit aufgenommen

wird, stehen dem gesamten Programm bzw. der Unit zur Verfügung, sind also global.

Per Definition stellt jede via **uses** aufgenommene Unit einen Block dar, der das folgende Programm und folgende angegebene Units umschließt. Die als erstes aufgeführte Unit entspricht folglich dem »äußersten« Block, die zuletzt aufgeführte Unit dem innersten Block. Daraus folgt: wenn zwei oder mehrere Units denselben Bezeichner deklarieren, beziehen sich »unqualifizierte« Referenzen (d.h. Angaben des Bezeichners ohne vorangestellten Unitnamen) auf den Bezeichner der zuletzt aufgeführten Unit – der Mechanismus ist also derselbe wie bei einer lokalen Redeklaration innerhalb des Programms.

Da die Unit *System*, in der alle vordefinierten Bezeichner von Turbo Pascal enthalten sind, implizit als allererste aufgenommen wird, können diese Bezeichner an jeder anderen Stelle redeklariert werden. Der Zugriff auf *System* ist über ein Voranstellen des Unitnamens dennoch jederzeit möglich: eine Angabe wie *System.WriteLine* ruft die vordefinierte Routine *WriteLn* auf – unabhängig davon, ob dieser Bezeichner an anderer Stelle redefiniert worden ist.

3. Typen

Jede Deklaration einer Variablen muß den Typ dieser Variablen angeben. Der Typ legt den Wertebereich der Variablen fest und bestimmt die Operationen, die mit ihr ausgeführt werden können. Eine Typen-Definition vereinbart also einen Bezeichner, der seinerseits für einen bestimmten Typ steht.

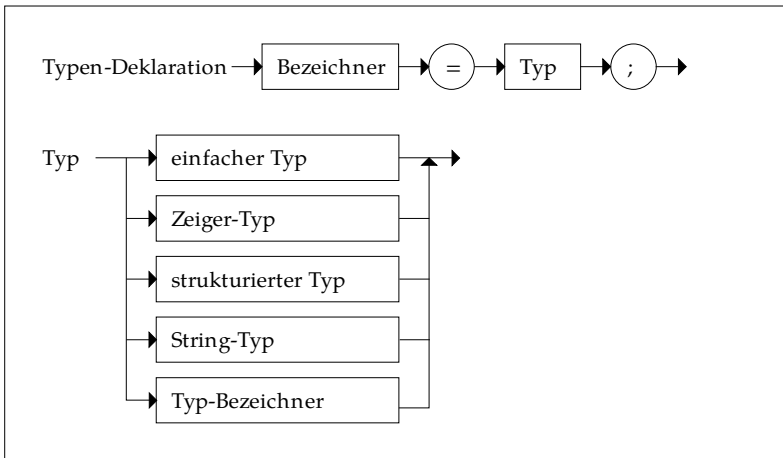


Abbildung 3.1 Syntax einer Typen-Definition

Wenn ein Bezeichner auf der linken Seite einer Typen-Definition steht, wird er für den Block, in dem seine Definition stattfindet, als Bezeichner eines Typs definiert. Der Gültigkeitsbereich eines Typen-Bezeichners erstreckt sich nicht auf die Definition selbst (d.h. die Definition kann den zukünftig definierten Typ nicht selbst enthalten). Die einzige Ausnahme von dieser Regel stellen Zeiger-Typen dar. Bezeichner werden je nach Art des Typs, für den sie stehen, in sechs Klassen unterteilt:

einfach	String
strukturiert	Zeiger
prozedural	Objekt

Einfache Typen

Einfache Typen definieren geordnete Mengen gleichartiger Werte.

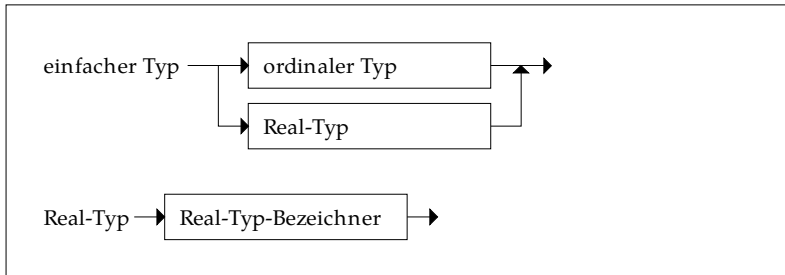


Abbildung 3.2 Syntax einfacher Typen

Der Bezeichner eines ganzzahligen Wertes steht für einen der fünf Integer-typen von Turbo Pascal. Der Bezeichner eines Realwertes hat den Typ *Real*, *Single*, *Double*, *Extended* oder *Comp*. Zur Notation von Integer-Konstanten und Real-Konstanten vergleichen Sie bitte die Abschnitte »Numerische Werte« und »Zeichenketten« in Kapitel 1 »Sprachelemente und Konstanten«.

Ordinale Typen

Ordinale (d.h. abzählbare) Typen stellen eine Untermenge der einfachen Typen dar. Alle einfachen Typen mit Ausnahme der Real-Typen sind abzählbar. Ordinaltypen unterscheiden sich von anderen Typen durch vier Merkmale:

- Alle möglichen Werte eines gegebenen Ordinal-Typs bilden eine geordnete Menge. Jeder mögliche Wert wird einer Ordinalzahl zugeordnet, die einen ganzzahligen Wert hat. Mit Ausnahme von Integer-Werten hat der erste Wert jedes Ordinal-Typs die Ordinalzahl 0, der nächste die Ordinalzahl 1 usw. Jeder mögliche Wert eines Ordinal-Typs lässt sich auf diese Weise zuordnen. In jedem Ordinal-Typ hat jeder Wert außer dem ersten einen Vorgänger und jeder außer dem letzten einen Nachfolger, die Reihenfolge richtet sich nach der Anordnung des Typs.
- Die Standard-Funktion *Ord* kann auf jeden ordinalen Wert angewendet werden und liefert die Ordinalzahl des betreffenden Wertes zurück.
- Die Standard-Funktion *Pred* kann auf alle Ordinal-Typen angewendet werden und liefert den Vorgänger des betreffenden Wertes zurück. Der Vorgänger des ersten Wertes eines Ordinal-Typs ist nicht definiert.
- Die Standard-Funktion *Succ* kann auf alle Ordinal-Typen angewendet werden und liefert den Nachfolger dieses Wertes zurück. Diese Funktion verhält sich

ähnlich wie *Pred*: Wenn sie auf den letzten Wert eines Ordinal-Typs angewendet wird, erzeugt auch sie einen Fehler.

Die Syntax eines Ordinal-Typs läßt sich folgendermaßen darstellen:

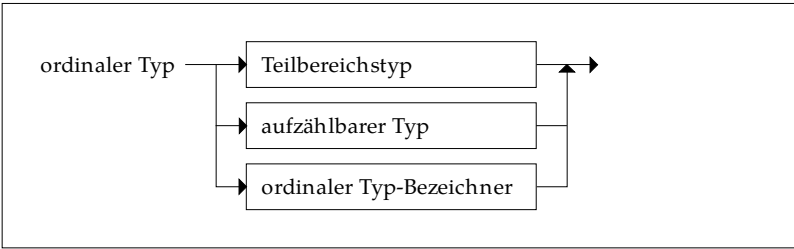


Abbildung 3.3 Syntax eines Ordinal-Typen

Turbo Pascal bietet sieben vordefinierte Ordinal-Typen: *ShortInt*, *Integer*, *LongInt*, *Byte*, *Word*, *Boolean* und *Char*. Zusätzlich sind zwei weitere Gruppen benutzerdefinierter Ordinal-Typen verfügbar: aufzählbare und Teilbereichs-typen.

Der Typ Integer

Integerwerte sind eine Teilmenge der ganzen Zahlen. In Turbo Pascal sind fünf verschiedene Typen vordefiniert, die jeweils unterschiedliche Wertebereiche haben:

Tabelle 3.1 Vordefinierte Integertypen und Wertebereiche

Typ	Bereich	Format
ShortInt	-128 .. 127	8 Bit mit Vorzeichen
Integer	-32768 .. 32767	16 Bit mit Vorzeichen
LongInt	-2147483648 .. 2147483647	32 Bit mit Vorzeichen
Byte	0 .. 255	8 Bit vorzeichenlos
Word	0 .. 65535	16 Bit vorzeichenlos

Arithmetische Operationen mit Integertypen erzeugen Resultate mit 8, 16 oder 32 Bit, wobei die folgenden Regeln gelten:

- Zur Speicherung von Konstanten wird der »kleinste« Typ benutzt, dessen Wertebereich den angegebenen Wert beinhaltet.
- Verknüpfungen zweier Integerwerte über einen binären Operator (d.h. Addition, Multiplikation usw.) erzwingen eine automatische Konvertierung beider Operanden in ein *gemeinsames Format*. Als gemeinsames Format zweier Operanden ist jeweils der »kleinste« Standardtyp definiert, dessen Wertebereich beide Operanden beinhaltet – das gemeinsame Format von *Byte* und *Integer* ist also *Integer*, das gemeinsame Format von *LongInt* und *Integer* ist *LongInt*. Die Operation wird mit dem gemeinsamen Format ausgeführt, das Ergebnis hat folglich ebenfalls dieses Format.

- Ausdrücke auf der rechten Seite einer Zuweisung werden unabhängig vom Format der Variablen auf der linken Seite durchgeführt.
- Werte, die ein einzelnes Byte Speicherplatz belegen (also *Char*, *Byte*, *ShortInt* und Aufzählungstypen bzw. entsprechende Teilbereiche) werden vor der Ausführung einer Operation vorzeichenrichtig auf 16 Bit erweitert und sind damit zu *Integer*- bzw. *Word*-Operanden kompatibel.

Ein Integerwert kann durch eine explizite Typ-Umwandlung jederzeit in ein anderes Integerformat überführt werden (siehe Kapitel 4 »Variablen« und 6 »Ausdrücke«).

Der Typ Boolean

Variablen vom Typ *Boolean* können nur einen der vordefinierten Werte *False* oder *True* annehmen. Da boolesche Werte als aufzählbarer Typ – nämlich als

```
type Boolean = (False, True);
```

definiert sind, gelten folgende Beziehungen:

- $False < True$
- $Ord(False) = 0$
- $Ord(True) = 1$
- $Succ(False) = True$
- $Pred(True) = False$
- $Succ(True)$ bzw. $Pred(False)$ sind nicht definiert

Der Typ Char

Die Grundmenge dieses Typs besteht aus Zeichen, deren Reihenfolge sich aus der Anordnung des IBM-Zeichensatzes bzw. der dazugehörigen ASCII-Codes ergibt. Der Aufruf der Funktion $Ord(Ch)$, bei dem *Ch* für einen Wert vom Typ *Char* steht, liefert die Ordinalzahl des Zeichens in *Ch* zurück.

Eine Stringkonstante, die nur aus einem Zeichen besteht, ist äquivalent zu einer Konstanten des Typs *Char*. Jeder Wert vom Typ *Char* kann über die Standardfunktion *Chr* erzeugt werden.

Aufzählbare Typen

Aufzählbare Typen definieren durch Aufzählung der Bezeichner geordnete Mengen. Die einzelnen Elemente erhalten Ordinalzahlen in der Reihenfolge ihrer Aufzählung: der erste Bezeichner erhält die Ordinalzahl 0, der zweite die Ordinalzahl 1 usw.

Wenn ein Bezeichner in der Bezeichnerliste eines aufzählbaren Typs erscheint, wird er in dem Block als Konstante deklariert, in dem auch der aufzählbare Typ deklariert wurde. Ein Beispiel für einen aufzählbaren Typ:

```
Karten = (Kreuz, Pik, Herz, Karo);
```

Durch diese Deklaration werden *Kreuz*, *Pik*, *Herz* und *Karo* zu Konstanten vom Typ *Karte*. Wenn die Funktion Ord auf einen aufzählbaren Wert angewendet

wird, liefert sie einen Integer zurück, der die Position des Wertes im Verhältnis zu den anderen Werten desselben aufzählbaren Typs bezeichnet. $Ord(Kreuz)$ liefert im vorangegangenen Beispiel also die Ordinalzahl 0 zurück, $Ord(Pik)$ die Ordinalzahl 1 usw.

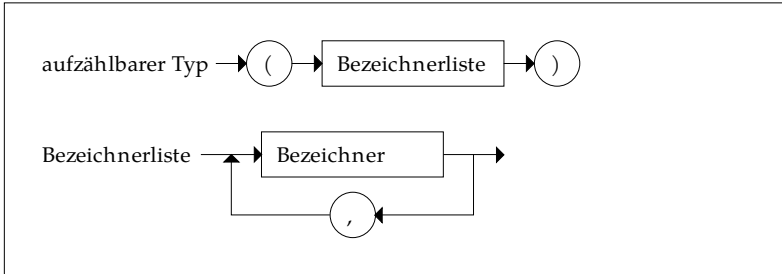


Abbildung 3.4 Syntax aufzählbarer Typen

Der Teilbereichstyp

Dieser Typ umfaßt einen ordinalen Wertebereich, der in diesem Zusammenhang auch als Grundmenge bezeichnet wird. Die Definition des Teilbereichstyps legt den jeweils niedrigsten und höchsten Wert dieses Bereichs fest. Die Syntax der Definition ist:

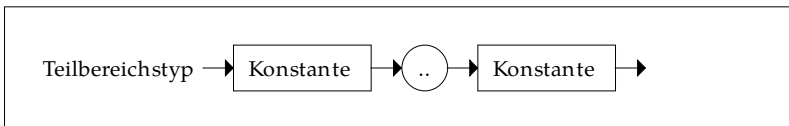


Abbildung 3.5 Syntax von Teilbereichstypen

Hier müssen beide Konstanten denselben Ordinaltyp haben. Bei einem Teilbereichstyp der Art $a..b$ muß a kleiner oder gleich b sein.

Beispiele für Teilbereichstypen sind:

`0..99`

`-128..127`

`'a'..'z'`

`Kreuz..Herz` {nach Definition des Typs Karte - s.o.}

Jede Variable eines Teilbereichstyps übernimmt die Eigenschaften der Grundmenge – ihr Wert muß sich allerdings zur Laufzeit in den durch den Teilbereich gesetzten Grenzen bewegen.

Die Sprachdefinition von Pascal legt fest, daß eine mit einer runden Klammer eingeleitete Wertefolge wie in

type

`Colors = (Red, Green, Blue);`

einen Aufzählungstyp definiert, und stellt den Compiler damit vor ein Problem, wenn ein Teilbereich über einen Konstanten-Ausdruck deklariert werden soll.

Bei einer Angabe wie

```
const
```

```
  RStart = 10;
```

```
  REnd   = 30;
```

```
type
```

```
  R_Scale = (REnd - RStart) * 2 .. (REnd + RStart) * 2;
```

ist die Absicht des Programmierers offensichtlich: es sollte ein Variablentyp definiert werden, der Werte von 40..80 speichern kann. Da sich der Compiler aber streng an die Sprachdefinition hält, erwartet er nach der öffnenden Klammer eine Folge von Wertangaben für einen Aufzählungstyp – und moniert folglich die beiden Punkte der Bereichsangabe als »unzulässig«.

Die Lösung ist recht einfach: Sie müssen lediglich dafür sorgen, daß der Ausdruck rechts des Gleichheitszeichens nicht mit einer Klammer *beginnt*:

```
type
```

```
  R_Scale = 2 * (REnd - RStart) .. (REnd + RStart) * 2;
```

Eine weitere Möglichkeit wäre, den ersten Ausdruck einer nur zu diesem Zweck definierten Konstanten zuzuweisen und diese Konstante dann für die Bereichsangabe zu verwenden, also:

```
const
```

```
  ...
```

```
  R_ScaleStart = (REnd - RStart) * 2;
```

```
type
```

```
  R_Scale = R_ScaleStart .. (REnd + RStart) * 2;
```

Was dieses Beispiel ebenfalls verdeutlicht: das »Klammerproblem« ist auf Typdefinitionen beschränkt – beim Festlegen von Konstanten-Werten macht es sich nicht bemerkbar.

Real- zahlen

Real-Typen sind eine Untermenge der reellen Zahlen und werden mit einer festgelegten Anzahl von Ziffern in Gleitkomma-Schreibweise dargestellt.

Ein Realwert n setzt sich normalerweise aus drei Komponenten zusammen: m , b und e . Hier gilt $m \cdot b^e = n$, wobei b immer den Wert 2 erhält und m sowie e ganzzahlige Werte im Bereich Real sind. Außerdem bestimmen die Werte m und e sowohl die Genauigkeit als auch den Bereich des Real-Typs.

Turbo Pascal kennt fünf verschiedene Arten von Real-Typen: *Real*, *Single*, *Double*, *Extended* und *Comp*. Sämtliche Typen außer *Real* sind nur dann verfügbar, wenn mit einem numerischen Coprozessor bzw. dem Emulator gearbeitet wird (siehe nächster Abschnitt). Die einzelnen Real-Typen unterscheiden sich voneinander sowohl durch ihren Wertebereich als auch in der Genauigkeit mit ihnen durchgeführter Operationen:

Tabelle 3.2 Bereich und Rechengenauigkeit von Real-Typen

Typ	Bereich	Genauigkeit
Real	$2.9 \cdot 10^{-39}$ bis $1.7 \cdot 10^{38}$	11-12 Stellen
Single	$1.5 \cdot 10^{-45}$ bis $3.4 \cdot 10^{38}$	7-8 Stellen
Double	$5.0 \cdot 10^{-324}$ bis $1.7 \cdot 10^{308}$	15-16 Stellen
Extended	$3.4 \cdot 10^{-4932}$ bis $1.1 \cdot 10^{4932}$	19-20 Stellen

Der Typ *Comp* umfaßt nur ganzzahlige Werte im Bereich von -2^{63} bis $2^{63}-1$, was ungefähr dem Bereich von $-9.2 \cdot 10^{18}$ bis $9.2 \cdot 10^{18}$ entspricht.

Turbo Pascal unterstützt zwei Arten der Code-Erzeugung für die Ausführung mathematischer Operationen mit Realzahlen, zwischen denen über den Compiler-Befehl *\$N* gewählt werden kann: *Emulation* (Berechnung über Bibliotheks-routinen) und *8087/80287* (Berechnung über einen numerischen Coprozessor bzw. Emulation eines Coprozessors).

Gleitkomma-Operationen über Bibliotheksroutinen

Der Schalter *\$N* ist als Standardvorgabe auf *\$N-* gesetzt, d.h. Gleitkomma-Operationen finden über Aufrufe der Laufzeitbibliothek statt. Aus Gründen der Arbeitsgeschwindigkeit und des Code-Umfangs sind bei dieser Vorgabe nur Operationen mit dem Format *Real* möglich – jeder Versuch der Compilierung eines Programms, das Berechnungen mit anderen Real-Formaten enthält, endet mit einer entsprechenden Fehlermeldung.

Gleitkomma-Operationen mit einem numerischen Coprozessor

Nach der Aktivierung des Schalters *\$N* (mit dem Compiler-Befehl *{ \$N+ }*) erzeugt der Compiler Code zur direkten Ansteuerung eines numerischen Coprozessors (8087, 80287 oder 80387). In diesem Modus sind sämtliche in Tabelle 3.2 aufgeführten Real-Formate für Berechnungen sowie ein weiterer Schalter verfügbar: *{ \$E+ }* legt die Aufnahme eines Emulators fest, über den ein Coprozessor notfalls simuliert wird; *{ \$E- }* bindet dagegen nur einige Routinen zur direkten Ansteuerung des 8087 in das Programm ein (und macht einen installierten Coprozessor damit zur Voraussetzung). Kapitel 13 »Der Coprozessor 8087« enthält weitere Details.

Der Typ String

Ein String-Typ ist eine Zeichenfolge mit dynamischer Länge, dem ein Speicherbereich konstanter Größe (1-255 Zeichen) zugewiesen ist. Die tatsächliche Länge des Strings zur Laufzeit des Programms wird dynamisch verwaltet. Wenn ein String-Typ ohne Größenzuweisung deklariert wird, bekommt er die Standardgröße 255 zugewiesen. Die Funktion *Length* liefert die aktuelle Länge eines Strings zurück.

Das Verhältnis zweier Stringwerte ergibt sich durch ihren Inhalt und durch ihre Länge: Zeichen auf gleicher Position innerhalb zweier Strings werden nach

ihrem ASCII-Code bewertet ('A' ist also kleiner als 'B', 'AA' ist kleiner als 'AB', 'Z' ist kleiner(!) als 'a'. Wenn zwei Strings bis zur Position n dieselben Zeichen enthalten, wird der kürzere von beiden als der kleinere bewertet ('Z' ist kleiner als 'Zs'). Ein *Nullstring*, d.h. ein String ohne Inhalt, stellt den kleinstmöglichen Wert dar.

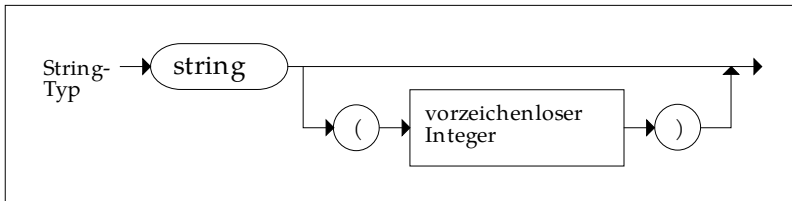


Abbildung 3.6 Syntax von String-Typen

In Kapitel 4 »Variablen« ist im Abschnitt »Arrays, Strings und Indizierungen« beschrieben, wie auf einzelne Zeichen eines Strings als Komponenten eines Arrays zugegriffen werden kann. Operatoren für String-Typen werden in Kapitel 6 »Ausdrücke« in den Abschnitten »String-Operatoren« und »Relationale Operatoren« erläutert; Erklärungen zu Stringprozeduren und -funktionen finden Sie im Kapitel 1 »Verzeichnis der Routinen« des *Referenzhandbuchs*.

Strukturierte Typen

Ein strukturierter Typ, der durch die Art seiner Strukturierung und durch seinen Komponenten-Typ bzw. seine Komponenten-Typen gekennzeichnet ist, enthält mehr als einen Wert. Wenn ein Komponenten-Typ strukturiert ist, hat der resultierende strukturierte Typ mehr als eine Strukturebene. Ein strukturierter Typ kann beliebig viele Strukturebenen haben.

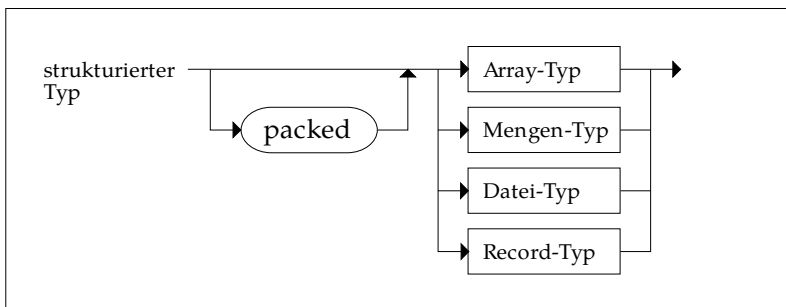


Abbildung 3.7 Syntax strukturierter Typen

Das reservierte Wort **packed** bei der Definition einer Datenstruktur weist den

Compiler an, diese Daten in »gepackter« Form (d.h. mit einem möglichst geringen Verbrauch an Speicherplatz) zu speichern – und das sogar dann, wenn dadurch die Zugriffsgeschwindigkeit auf eine Variable dieses Typs herabgesetzt wird. Da Turbo Pascal grundsätzlich mit gepackten Formaten arbeitet, wird **packed** lediglich formalsyntaktisch bei der Compilierung geprüft – eine weitere Wirkung hat es nicht.

Hinweis: Turbo Pascal erlaubt strukturierte Typen mit einem maximalen Umfang von 65520 Bytes.

Array-Typen

Arrays haben eine festgelegte Anzahl von Komponenten eines einzigen Typs, dem Komponenten-Typ. Im Syntaxdiagramm für Arrays folgt der Komponenten-Typ auf das reservierte Wort **of**.

Die Anzahl der Elemente jeder Dimension wird über den jeweiligen Index-Typ festgelegt, wobei alle ordinalen Grundtypen zulässig sind, außer dem Typ *Longint* und Teilbereichen von *Longint*. Die Grenzen von Teilbereichstypen müssen sich innerhalb des Wertebereichs einfacher Ordinal-Variablen bewegen: $[0..MaxInt]$ stellt einen zulässigen Index-Typ dar, $[65000..75000]$ dagegen nicht.

Ein Array kann in jeder Dimension über alle Werte seines Index-Typs indiziert werden; die Maximalzahl der Elemente einer Dimension ist also mit der Anzahl der möglichen Werte des Index-Typs identisch.

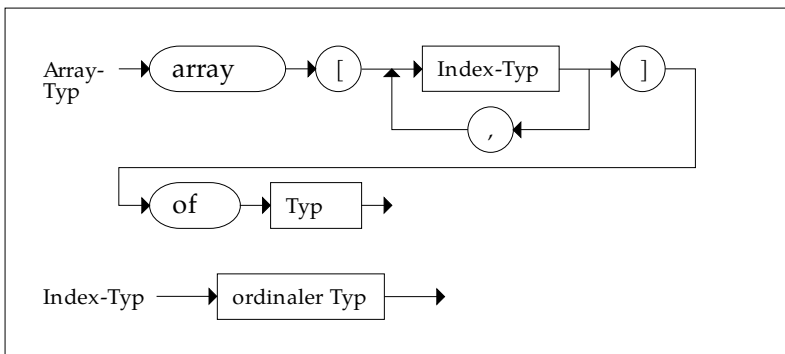


Abbildung 3.8 Syntax von Array- und Index-Typen

Ein Beispiel für einen Array-Typ:

array[1..100] of Real

Wenn der Komponenten-Typ eines Arrays ebenfalls ein Array ist, kann man das Ergebnis entweder als ein Array **of** Array oder als ein einziges mehrdimensionales Array behandeln. Die Deklaration

array[Boolean] of array[1..10] of array[Anzahl] of Real

wird vom Compiler genauso interpretiert wie

array[Boolean, 1..10, Anzahl] of Real

Aus demselben Grund sind die beiden folgenden Deklarationen identisch:

```
packed array[1..10] of packed array[1..8] of Boolean
packed array[1..10, 1..8] of Boolean
```

Der Zugriff auf Komponenten eines Arrays geschieht durch Anfügen eines oder mehrerer Indizes in eckigen Klammern hinter dem Arraynamen (siehe Abschnitt »Arrays, Strings und Indizierungen« in Kapitel 4 »Variablen«). Wenn m einen kleineren Wert als n hat, dann wird ein Array-Typ der Form

```
packed array[m..n] of Char
```

als gepackter String bezeichnet (das reservierte Wort **packed** könnte in Turbo Pascal auch fehlen, weil sowieso »gepackt« wird). Dieser Typ hat einige Eigenschaften, über die die anderen Array-Typen nicht verfügen. Näheres dazu finden Sie im Abschnitt »Identische und kompatible Typen« in diesem Kapitel.

Record-Typen

Ein Record-Typ beinhaltet eine festgelegte Anzahl von Komponenten oder Feldern, die ihrerseits verschiedene Typen haben können. Die Definition eines Record-Typs legt den Typ und den Bezeichner jedes Feldes fest.

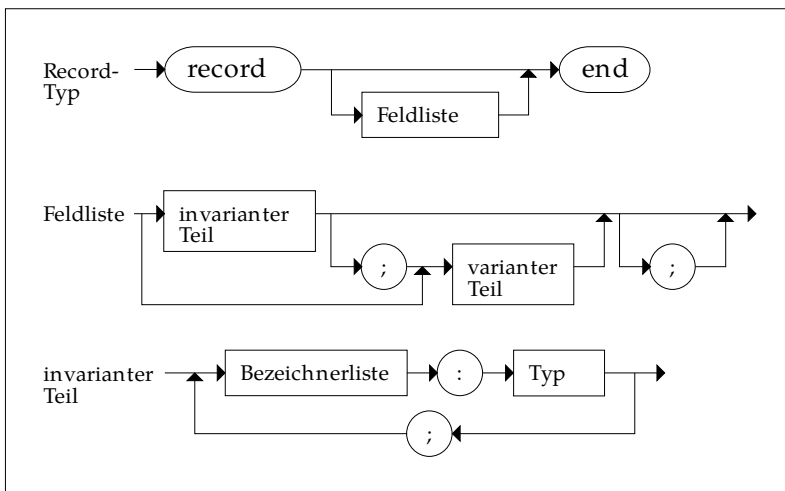


Abbildung 3.9 Syntax von Record-Typen

Der festgelegte Teil eines Record-Typen definiert eine Folge von Feldern, wobei jedes Feld einen Bezeichner und einen Typ erhält. Die Inhalte dieser Felder werden immer auf die gleiche Weise gelesen. Die folgenden Zeilen definieren einen Record-Typ des Namens *Datum* mit festgelegten Feldern:

type

```
Datum = record
  Jahr: Integer;
  Monat: 1..12;
```

```

Tag: 1..31;
end;

```

Der variante Teil eines Records definiert Felder, die mit Werten unterschiedlichen Typs belegt werden können: ein Feld kann beispielsweise entweder einen booleschen Wert, einen Integer oder einen String enthalten. Der Compiler weist bei varianten Feldern jeweils automatisch Speicherplatz für das größte der aufgeführten Elemente zu. Auf variante Felder kann auf mehrere Arten zugegriffen werden – je nachdem, welchen Inhalt sie zum Zeitpunkt des Zugriffs haben.

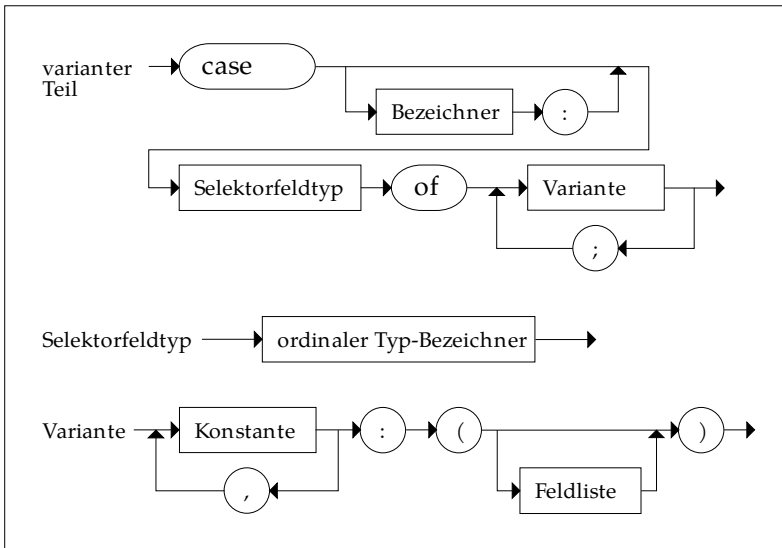


Abbildung 3.10 Syntax varianter Teile eines Records

Wie das Syntaxdiagramm verdeutlicht, wird jede Variante durch mindestens eine Konstante bezeichnet. Alle Konstanten müssen eindeutig sein und einen Ordinal-Typ haben, der zum Typ des Selektorfeldes kompatibel ist. Auf Varianten wird auf dieselbe Art zugegriffen wie auf festgelegte Felder.

Durch die Definition eines weiteren (optionalen) Bezeichners im varianten Teil der Feldliste läßt sich ein *Selektorfeld-Bezeichner* festlegen, der damit zum Bezeichner eines zusätzlichen Feldes des Records – des *Selektorfeldes* – wird. Das Programm kann den Wert dieses Feldes benutzen, um zu bestimmen, welche Variante zu einem bestimmten Zeitpunkt gerade aktiv ist. Ist kein Selektorfeld vorhanden, muß das Programm die Art einer aktiven Variante über andere Kriterien bestimmen. Die folgenden Beispiele stellen gültige Definitionen varianter Records dar:

```

record
  x, y: Real;
  case Art: Umriss of
    Rechteck: (Hoehe, Breite: Real);

```

```

        Dreieck:  (Groesse1, Groesse2, Winkel: Real);
        Kreis:    (Radius: Real);
end;
record
    Vorname, Nachname: string[40];
    Geburtsdatum: Datum;
    case Staatsbuerger: Boolean of
        true:  (Geburtsort: string[40];
        false: (Geburtsland: string[20];
                Einreiseort: string[20];
                Einreisedatum: Datum;
                Ausreisedatum: Datum;
    end;

```

Objekttypen

Ein Objekttyp ist eine Struktur, die aus einer bestimmten Anzahl von Komponenten besteht. Jede dieser Komponenten ist entweder ein *Feld*, das Daten eines bestimmten Typs enthält, oder eine *Methode*, die eine Operation mit dem Objekt durchführt. Entsprechend einer Variablendeklaration, gibt man in der Deklaration eines Felds den Datentyp des Felds und eine Bezeichnung für das Feld an. Und entsprechend einer Prozedur- oder Funktionsdeklaration enthält die Deklaration einer Methode den Kopf einer Prozedur, einer Funktion, eines Konstruktors oder eines Destruktors.

Ein Objekttyp kann Komponenten von einem anderen Objekttyp *erben*. Wenn T_2 die Eigenschaften von T_1 übernimmt, dann ist T_2 ein *Nachkomme* von T_1 und T_1 ist ein *Vorfahr* von T_2 .

Die Vererbung ist ein transitiver Vorgang. Das bedeutet, daß T_3 auch die Eigenschaften von T_1 erbt, wenn T_3 das Objekt T_2 und T_2 das Objekt T_1 erbt. Der *Bereich* eines Objekttyps umfaßt das Objekt selbst und alle seine Nachkommen.

Es folgen einige Beispiele für die Deklaration von Objekten. Diese Deklarationen werden im weiteren Verlauf dieses Kapitels immer wieder verwendet:

```

type
    Point = object
        X, Y : Integer;
    end;

    Rect = object
        A, B : Point;
        procedure Init(XA, YA, XB, YB :Integer);
        procedure Copy(var R : Rect);
        procedure Move(DX, DY : Integer);
        procedure Grow(DX, DY : Integer);
        procedure Intersect(var R : Rect);
        procedure Union(var R : Rect);
        function Contains(P : Punkt) : Boolean;
    end;

```

```

StringPtr = ^String;
FieldPtr = ^Field;
Field = object
  X, Y, Len : Integer;
  Name : StringPtr;
  constructor Copy(var F : Field);
  constructor Init(FX, FY, FLen: Integer; FName:String);
  destructor Done; virtual;
  procedure Display; virtual;
  procedure Edit; virtual;
  function GetStr : String; virtual;
  function PutStr(S : String) : Boolean; virtual;
end;

StrFieldPtr = ^StrField;
StrField = object (Field)
  Value : StringPtr;
  constructor Init(FX, FY, FLen : Integer; FName : String);
  destructor Done; virtual;
  function GetStr : String; virtual;
  function PutStr (S : String) : Boolean; virtual;
  function Get : String;
  procedure Put(S : String);
end;

NumFieldPtr = ^NumField;
NumField = object (Field)
  Value, Min, Max : Longint;
  constructor Init(FX, FY, FLen : Integer; FName : String;
    FMin, FMax : Longint);
  function GetStr : String; virtual;
  function PutStr (S : String) : Boolean; virtual;
  function Get : Longint;
  procedure Put(N : Longint);
end;

ZipFeldPtr = ^ZipField;
ZipField = object (Field)
  function GetStr : String; virtual;
  function PutStr(S : String) : Boolean; virtual;
end;

```

Im Gegensatz zu anderen Datentypen kann ein Objekttyp nur in einer Typdeklaration des äußersten Gültigkeitsbereichs eines Programms oder einer Unit deklariert werden. Ein Objekttyp darf also nicht im Variablenabschnitt oder in einer Prozedur, Funktion oder Methode deklariert werden.

Unter den Komponenten eines Datentyps darf kein Objekttyp und keine Struktur mit einem Objekttyp als Komponente angegeben werden.

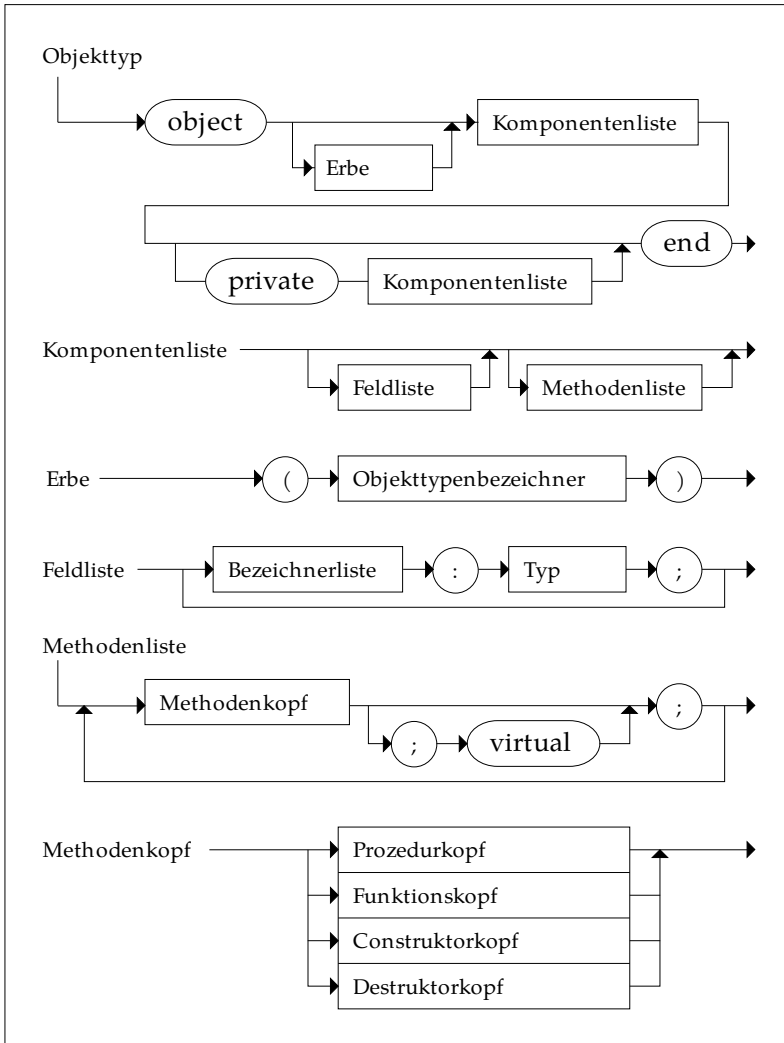


Abbildung 3.11 Syntaxerweiterungen der objektorientierten Programmierung

Der Gültigkeitsbereich eines Komponentenbezeichners umfaßt den gesamten Bereich eines Objekttyps. Darüber hinaus bezieht sich der Gültigkeitsbereich eines Komponentenbezeichners auf alle Prozedur-, Funktions-, Konstruktor- und Destruktorböcke, in denen die Methoden des Objekttyps und seiner Nachkommen definiert sind. Aus diesem Grund muß ein Komponentenbezeichner innerhalb eines Objekttyps sowie dessen Nachkommen und in all seinen Methoden eindeutig bezeichnet sein.

Der Gültigkeitsbereich eines im **private**-Abschnitt deklarierten Bezeichners

beschränkt sich auf das Modul (Programm oder Unit), das die entsprechende Objekttyp-Deklaration enthält. Das heißt, **private** Komponenten-Bezeichner verhalten sich wie normale öffentliche (*public*) Komponenten-Bezeichner innerhalb des Moduls, das die Objekttyp-Deklaration enthält. Außerhalb des Moduls sind **private** Komponenten-Bezeichner nicht definiert. Durch Deklaration verwandter Objekttypen innerhalb desselben Moduls erhalten diese Objekttypen Zugriff auf sämtliche Komponenten der anderen Objekttypen, ohne die **private**-Komponenten anderen Modulen sichtbar zu machen.

Die Deklaration einer Methode innerhalb eines Objekttyps entspricht einer **forward**-Deklaration dieser Methode. Die Methode muß also irgendwann hinter der Deklaration des Objekttyps innerhalb desselben Gültigkeitsbereichs wie die Deklaration des Objekttyps in Form einer Definition *implementiert* werden.

Wenn eine spezifische Unterscheidung erforderlich ist, muß der vollständige Methodenbezeichner angegeben werden. Dieser setzt sich aus dem Bezeichner des Objekttyps, einem dahinter stehenden Punkt (.) und dem Bezeichner der Methode zusammen. Wie bei jedem anderen Bezeichner kann vor dem vollständigen Methodenbezeichner noch ein Unit-Bezeichner und ein weiterer Punkt vorangestellt werden, falls dies erforderlich ist.

Innerhalb der Deklaration eines Objekttyps kann man in der Kopfzeile einer Methode die Parameter des deklarierten Objekttyps angeben, auch wenn die Deklaration noch nicht abgeschlossen ist. Dies wurde in den Methoden *Copy*, *Intersect* und *Union* des Typs *Rect* im vorherigen Beispiel veranschaulicht.

Methoden sind in der Standardeinstellung *statisch*, können aber mit Ausnahme der Konstruktorfunktionen durch die Anweisung **virtual** in der Deklaration der Methode als *virtuell* deklariert werden. Der Compiler löst die Aufrufe von statischen Methoden zur Compilationszeit auf, während die virtuellen Methoden zur Laufzeit ermittelt werden. Letzteres wird auch als *dynamische Bindung* bezeichnet.

Wenn in einem Objekttyp virtuelle Methoden deklariert oder vererbt werden, müssen die Variablen dieses Typs mit einem Aufruf eines Konstruktors *initialisiert* werden, bevor die virtuelle Methode aufgerufen werden kann. In jedem Objekttyp, in dem virtuelle Methoden deklariert oder vererbt werden, muß mindestens ein Konstruktor deklariert oder vererbt werden.

Ein Objekttyp kann jede der Methoden, die von einem Vorfahren vererbt werden, *ersetzen* (neu definieren). Wenn bei der Deklaration einer Methode in einem Nachkommen dieselbe Bezeichnung verwendet wird wie bei einem Vorfahren, dann ersetzt die Deklaration im Nachkommen die Deklaration des Vorfahren. Der Gültigkeitsbereich der neuen Methode umfaßt den Bereich, in dem der Nachkomme eingeführt wurde und erstreckt sich auch auf weitere Nachkommen, es sei denn, diese ersetzen ihrerseits die betreffende Methode.

In der Neudefinition einer statischen Methode kann man die Kopfzeile der Methode beliebig angeben. Im Gegensatz dazu muß die Neudefinition einer virtuellen Methode in Bezug auf Reihenfolge, Typ und Bezeichnung der Parameter sowie dem Typ des Funktionsergebnisses, falls eine Funktion vorliegt, exakt übereinstimmen. Darüber hinaus muß in der Neudefinition wieder die Anweisung **virtual** stehen.

Ein Objekt wird *eingesetzt*, in einer »Instanz« erzeugt, indem man eine Variable oder eine typisierte Konstante eines Objekttyps deklariert oder die

Standardprozedur *New* auf eine Zeigervariable eines Objekttyps anwendet. Das Ergebnis wird als *Instanz* des Objekttyps bezeichnet.

```
var
  F : Field;
  Z : ZipField;
  FP : FieldPtr;
  ZP : ZipFieldPtr;
```

In diesen Variablendeklarationen ist *F* eine Instanz von *Field*, *Z* eine Instanz von *ZipField*. In gleicher Weise zeigen *FP* und *ZP* nach dem Aufruf von *New* mit *FP* und *ZP* als Parameter auf Instanzen von *Field* und *ZipField*.

Wenn ein Objekttyp virtuelle Methoden enthält, dann müssen die Instanzen dieses Objekttyps mit Hilfe eines Konstruktor-Aufrufs initialisiert werden, bevor eine der virtuellen Methoden aufgerufen werden kann. Hierzu ein Beispiel:

```
var
  S: StrField;

begin
  S.Init(1, 1, 25, 'Vorname');
  S.Put('Frank');
  S.Display;
  ...
  S.Done;
end;
```

Würde *S.Init* nicht aufgerufen, führte der Aufruf von *S.Display* zu einer undefinierten Adresse. Der Aufruf *S.Put* jedoch ist eindeutig, da *Put* eine statische Methode von *StrField* ist.

Hinweis: Die Zuweisung zu der Instanz eines Objekttyps initialisiert die Instanz *nicht*.

Natürlich müssen auch solche Instanzen initialisiert werden, die Bestandteil eines strukturierten Typs sind. Zum Beispiel:

```
var
  Comment: array[1..5] of StrField;
  i: Integer;

begin
  for i:= 1 to 5 do Comment[i].Init(1, i+10, 40, 'Kommentar');
  ...
  for i:= 1 to 5 do Comment[i].Done;
end;
```

Bei dynamischen Instanzen ist in den meisten Fällen die Initialisierung mit einer Speicherplatzreservierung verbunden. Beim Aufräumen wird gleichzeitig der reservierte Speicherplatz freigegeben. Für die Heap-Verwaltung wird gewöhnlich, wie im folgenden Beispiel, die erweiterte Syntax der Standard-Prozeduren *New* und *Dispose* gebraucht:

```

var
  SP: StrFieldPtr;

begin
  New(SP, Init(1, 1, 25, 'Vorname'));
  SP^.Put('Frank');
  SP^.Display;
  ...
  Dispose(SP, Done);
end;

```

Ein Zeiger auf einen Objekttyp kann einem Zeiger eines Vorfahren zugewiesen werden. Deshalb kann ein Zeiger auf einen Objekttyp zur Laufzeit eines Programms auf eine Instanz dieses Typs oder dessen Vorfahren zeigen.

Zum Beispiel kann man einen Zeiger des Typs *ZipFieldPtr* an Zeiger des Typs *ZipFieldPtr*, *NumFieldPtr* und *FieldPtr* zuweisen. Während der Programmausführung kann ein Zeiger des Typs *FieldPtr* entweder auf *nil* oder auf eine Instanz von *Field*, *StrField*, *NumField* oder *ZipField* beziehungsweise jede andere Instanz eines Nachkommens von *Field* zeigen.

Diese Kompatibilität bei Zeigerzuweisungen gilt auch für *var*-Parameter, die als Objekttyp deklariert sind. Zum Beispiel kann die Methode *Field.Copy* an eine Instanz von *Field*, *StrField*, *NumField*, *ZipField* oder jede andere Instanz eines Nachkommens von *Field* zugewiesen werden.

Eine Methode wird über einen Bezeichner der Form *Instanz.Methode* aufgerufen, wobei *Instanz* eine Instanz eines Objekttyps angibt und *Methode* eine Funktion des entsprechenden Objekttyps ist.

Bei den statischen Methoden bestimmt der (zur Compilationszeit) *deklarierte* Typ von *Instanz*, welche Methode aktiviert wird. Zum Beispiel aktivieren die Bezeichner *F.Init* und *FP.Init* immer *Field.Init*, da der deklarierte Typ von *F* und *FP* in beiden Fällen *Field* ist.

Bei den virtuellen Methoden steuert der (zur Laufzeit) *tatsächlich vorliegende* Typ die Auswahl. Zum Beispiel kann der Bezeichner *FP.Edit* die Methoden *Field.Edit*, *StrField.Edit*, *NumField.Edit* oder *ZIPField.Edit* aktivieren. Welche Methode tatsächlich aktiviert wird, hängt von der jeweiligen Instanz ab, auf die *FP* zeigt.

Im allgemeinen gibt es keine Möglichkeit, um feststellen zu können, welche Methode von einem virtuellen Methodenbezeichner aktiviert werden kann. Sie können zum Beispiel eine Routine entwickeln (beispielsweise eine Eingaberoutine für einen Maskeneditor), die *FP.Edit* aktiviert und später, ohne die Routine zu verändern, auf einen noch nicht vorhergesehenen Nachkommen von *Field* anwenden. Wenn eine Erweiterung dieser Art gewünscht wird, sollten Sie einen Objekttyp verwenden, der beliebig viele Typen an Nachkommen zulässt, anstatt einen Record-Typ mit einer begrenzten Anzahl an Varianten einzusetzen.

Mengen-Typen

Der Wertebereich eines Mengen-Typs ist die Menge eines bestimmten Ordinal-Typs, des *Grundtyps* der Menge. Jeder mögliche Wert eines Mengen-Typs ist eine Teilmenge der möglichen Werte dieses Grundtyps.

Eine Variable des Typs *Menge* kann eine beliebige Anzahl der Elemente des

Grundtyps enthalten, die Definition einer leeren Menge (die kein einziges Element enthält) ist ebenfalls möglich.

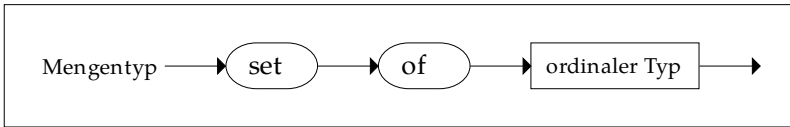


Abbildung 3.12 Syntax von Mengen-Typen

Der Grundtyp einer Menge darf nicht mehr als 256 mögliche Werte beinhalten; die niedrigste und die höchste Ordinalzahl des Typs müssen im Bereich von 0..255 liegen. Aus diesem Grund sind die Typen *ShortInt*, *Integer*, *LongInt* und *Word* nicht als Grundtyp einer Menge zulässig.

Operatoren für Mengen-Typen sind in Kapitel 6 »Ausdrücke« im Abschnitt »Mengen-Operatoren« erläutert. Der Abschnitt »Mengenkonstruktoren« desselben Kapitels erklärt, wie Mengen definiert werden. Jeder Mengen-Typ kann den Wert `[]` enthalten, die sogenannte *leere Menge*.

Dateitypen

Ein Dateityp besteht aus einer linearen Folge von Komponenten eines Typs, dem Komponenten-Typ. Er kann jeden beliebigen Typ haben (außer den Dateityp selber oder einen strukturierten Typ mit einer Komponente des Typs *File*). Die Anzahl der Komponenten wird durch die Deklaration des Dateityps nicht definiert.

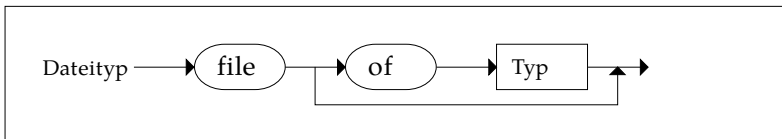


Abbildung 3.13 Syntax von Dateitypen

Wenn das reservierte Wort **of** und der Komponenten-Typ nicht angegeben sind, wird die Datei als *untypisiert* bezeichnet. Derartige Dateien werden vorwiegend für Ein- und Ausgaben auf niedriger Ebene verwendet.

Der Standard-Dateityp *Text* bezeichnet eine Datei mit zeilenweise angeordneten Zeichenfolgen. Textdateien benutzen besondere Prozeduren zur Ein- und Ausgabe, die in Kapitel 10 »Ein- und Ausgabe« erläutert werden.

Zeiger-Typen

Ein Zeiger-Typ (Pointer-Typ) definiert eine Menge von Werten, die auf dynamische Variablen eines festgelegten Typs (*Grundtyp*) zeigen. Eine Variable

dieses Typs enthält die Speicheradresse eines anderen Objekts (bei dem es sich in den meisten Fällen um eine dynamische erzeugte Variable handeln wird).

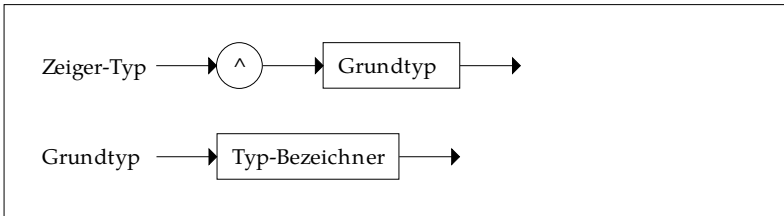


Abbildung 3.14 Syntax von Zeiger-Typen

Wenn der Grundtyp ein noch nicht deklarierter Bezeichner ist, muß er im selben Deklarationsteil wie der Zeiger-Typ vereinbart werden. Mit der Prozedur *New*, der Funktion *Ptr* oder dem Adreß-Operator *@* läßt sich einer Zeiger-Variablen ein Wert zuweisen. Die Prozedur *New* weist einer dynamischen Variablen einen neuen Speicherbereich (im Heap-Bereich für Anwendungen) zu und speichert die Adresse dieser Variablen in der Zeiger-Variablen. Der Operator *@* setzt die Zeiger-Variable auf die Startadresse eines Speicherbereichs, der existierende Variablen enthält – hier spielt es keine Rolle, ob diese Variablen bereits Bezeichner haben oder nicht. Die Funktion *Ptr* setzt die Zeiger-Variable auf einen bestimmten Wert, d.h. eine bestimmte Speicheradresse.

Das reservierte Wort **nil** (»not in list« = nicht in der Liste) steht für einen Zeigerwert, der *nirgendwohin* zeigt (vgl. Kapitel 4 »Variablen«).

Der Standard-Typ *Pointer* steht für einen *untypisierten* Zeiger, d.h. einen Zeiger, der auf keinen bestimmten Variablentyp zeigt. Zeigervariablen dieses Typs sind wie die Konstante **nil** mit allen anderen Zeigern kompatibel – da eine als *Pointer* deklarierte Zeigervariable *p* aber auf keinen bestimmten Variablentyp zeigt, werden Bezüge der Form $p^{\wedge}$ (also die Ermittlung einer Adresse) als Fehler gewertet.

Prozedur-Typen

Als Erweiterung des Sprachstandards ermöglicht Turbo Pascal die Behandlung von Prozeduren und Funktionen als *Objekte* des Programms (und nicht als unveränderlicher Teil von ihm). Eine neue Klasse von Variablen – die *Prozedur-Variablen* – speichert nicht nur (wie Zeiger) die Adresse einer Routine, sondern hat für den Compiler auch die restlichen Charakteristika einer Prozedur bzw. Funktion (Parameterzahl, -typen und -folge sowie Ergebnistypen von Funktionen). Abgesehen von diesen speziellen Eigenschaften sind Prozedur-Variablen normale Daten, können deshalb auch als Parameter übergeben werden und ermöglichen es damit, einer Routine programmgesteuert mitzuteilen, welche anderen Routinen sie aufrufen soll.

Zusammenhängen obligatorisch ist. Für andere Operationen reicht eine *Kompatibilität* der beteiligten Typen aus (wie z.B. *Integer* und *LongInt*). Für Zuweisungen gelten spezielle Anforderungen – hier spricht man von *Zuweisungs-Kompatibilität* (z.B. bei der Zuweisung eines Integerwertes an eine Real-Variable).

Zwei Typen sind identisch, wenn sie entweder mit demselben Typ-Bezeichner deklariert worden sind oder ihre Definition sich aus demselben Bezeichner ableitet.

Identische Typen

Die Gleichheit zweier Typen ist nur in Prozedur- und Funktionsaufrufen und auch in diesem Fall lediglich zwischen aktuellen und formalen Variablen-Parametern gefordert.

Die beiden Typen T_1 und T_2 sind identisch, wenn eine der folgenden Bedingungen erfüllt ist: T_1 und T_2 sind derselbe Typen-Bezeichner; T_1 ist als äquivalent zu einem Typ deklariert, der mit T_2 identisch ist.

Die zweite Bedingung läuft darauf hinaus, daß T_1 nicht direkt als äquivalent zu T_2 deklariert werden muß. Aus einer Typen-Definition wie

```
T1 = Char ;
T2 = T1 ;
T3 = Char ;
T4 = T2 ;
```

ergeben sich T_1, T_2, T_3, T_4 und *Char* als identische Typen. Die Vereinbarungen

```
T5 = set of Char ;
T6 = set of Char ;
```

vereinbaren T_5 und T_6 nicht als identische Typen, weil **set of Char** kein Typ-Bezeichner ist. Zwei Variablen, die in derselben Deklaration stehen, haben dagegen auch den gleichen Typ:

```
V1, V2 : set of Char ;
```

Bei einer getrennten Deklaration ist das dagegen nicht der Fall:

```
V1 : set of Char ;
V2 : set of Char ;
V3 : Char ;
V4 : Char ;
```

Hier haben nur V_3 und V_4 denselben Typ, nicht aber V_1 und V_2 .

Kompatibilität von Typen

Kompatibilität zwischen zwei Typen kann beispielsweise bei Ausdrücken oder auch bei relationalen Operationen erforderlich sein. Darüber hinaus ist die Kompatibilität zwischen Typen die Voraussetzung zur Kompatibilität von Zuweisungen. Zwei Typen sind kompatibel, wenn zumindest eine der folgenden Bedingungen erfüllt ist:

- beide Typen sind gleich;
- beide Typen sind Real-Typen;

- beide Typen sind Integer-Typen;
- ein Typ ist ein Teilbereich des anderen;
- beide Typen sind Teilbereiche derselben Grundmenge;
- beide Typen sind Mengen-Typen mit kompatiblen Grundtypen;
- beide Typen sind gepackte String-Typen mit der gleichen Anzahl von Komponenten;
- ein Typ ist ein String-Typ und der andere ist ein String-Typ, ein gepackter String-Typ oder ein Char-Typ;
- ein Typ ist der vordefinierte Typ *Pointer*, der andere ein beliebiger Zeiger-Typ
- beide Typen sind prozedurale Typen mit identischen Ergebnistypen sowie gleicher Zahl und Reihenfolge von Parametern identischen Typs.

Zuweisungs-Kompatibilität

Diese Art von Kompatibilität ist unabdingbar, wenn ein Wert zugewiesen werden soll (z.B. bei einer direkten Zuweisung mit $:=$ oder bei einer Parameterübergabe). Der Wert eines Typs T_2 kann einem Wert T_1 zugewiesen werden (d.h. $T_1 := T_2$), wenn eine der folgenden Bedingungen erfüllt ist:

- T_1 und T_2 sind vom gleichen Typ; keiner von beiden ist ein Dateityp oder ein strukturierter Typ, der als Komponente einen Dateityp enthält;
- T_1 und T_2 sind kompatible Ordinal-Typen; der Wert von T_2 liegt im möglichen Wertebereich von T_1 ;
- T_1 und T_2 sind Real-Typen; der Wert von T_2 fällt in den möglichen Wertebereich von T_1 ;
- T_1 hat den Typ Real, T_2 den Typ Integer;
- T_1 und T_2 sind String-Typen;
- T_1 ist ein String-Typ, T_2 ein Char-Typ;
- T_1 und T_2 sind gepackte String-Typen;
- T_1 ist ein String-Typ, T_2 ein gepackter (*packed*) String-Typ;
- T_1 und T_2 sind kompatible Mengen-Typen; alle Elemente des Wertes von T_2 fallen in den möglichen Wertebereich von T_1 ;
- T_1 und T_2 sind kompatible Zeiger-Typen;
- T_1 und T_2 sind kompatible prozedurale Typen;
- T_1 und T_2 sind prozedurale Typen mit identischen Ergebnistypen sowie gleicher Zahl und Reihenfolge von Parametern identischen Typs;
- Ein Objekttyp T_2 ist zuweisungskompatibel mit einem Objekttyp T_1 , wenn T_2 im Bereich von T_1 liegt;
- Ein Zeigertyp P_2 , der auf einen Objekttyp T_2 zeigt, ist zuweisungskompatibel

zu einem Zeigertyp P_2 , der auf einen Objekttyp T_1 zeigt, wenn T_2 im Bereich von T_1 liegt.

Wenn keine dieser Bedingungen erfüllt, Zuweisungs-Kompatibilität aber erforderlich ist, meldet entweder der Compiler einen Fehler oder die Programmausführung bricht an der entsprechenden Stelle ab.

Der Typen-Definitionsteil

Alle Programme, Prozeduren oder Funktionen, in denen Typen definiert werden, müssen zu diesem Zweck einen Typen-Definitionsteil haben. Dieser Teil könnte beispielsweise so aussehen:

type

```

Bereich      = Integer;
Zahl         = Integer;
Farbe        = (rot, gruen, blau);
CharVal      = Ord('A')..Ord('Z');    { ein berechneter Bereich }
TestIndex    = 1..100;
Testwert     = -99..99;
TestList     = array[TestIndex] of Testwert;
TestListPtr  = ^TestList;
Datum        = object
    Jahr : Integer;
    Monat : 1..12;
    Tag : 1..31;
    procedure SetzeTag(T, M, J: Integer);
    function ZeigeDatum: String;
end;

Messdaten    = record
    Wann : Datum;
    Anzahl : TestIndex;
    Daten : TestListPtr;
end;

Messliste    = array[1..50] of Messdaten;
Name         = string[80];
Geschlecht   = (weiblich, maennlich);
Person       = ^PersonDaten;
PersonDaten  = record
    Name, Vorname : Name;
    Alter : Integer;
    verheiratet : Boolean;
    Vater, Kind, Geschwister : Person;
    case s: Geschlecht of
        maennlich : (baertig : Boolean);
        weiblich : (schwanger : Boolean);

```

```
                end;  
PersonBuffer   = array[0..SizeOf(PersonDaten)-1] of Byte;  
Leute          = file of PersonDaten;
```

In diesem Beispiel sind *Bereich*, *Zahl* und *Integer* identische Typen. *TestIndex* ist zu diesen drei Typen kompatibel *und* zuweisungskompatibel – nicht aber identisch.

Hinweis: Der zulässige Bereich von *CharVal* und die Arraygröße des Typs *PersonBuffer* wurden hier mit Hilfe berechneter Konstanten definiert.

4. Variablen

Die Deklaration von Variablen

Eine Variablen-Deklaration enthält eine Liste von Bezeichnern, die ihrerseits für neue Variablen und ihren jeweiligen Typ stehen.

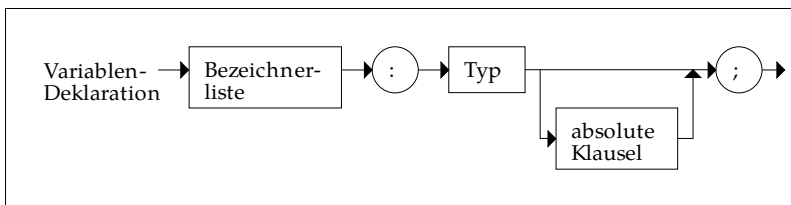


Abbildung 4.1 Syntax einer Variablen-Deklaration

Neben der Möglichkeit, einen neuen Typ innerhalb einer Variablen-Deklaration zu vereinbaren, lässt sich auch ein Typ zuweisen, dessen Typ-Bezeichner bereits vorher im Deklarationsteil desselben Blocks, in einem übergeordneten Block oder einer Unit vereinbart worden ist. Beispiele für gültige Variablen-Deklarationen sind:

var

```

x, y, z: Real;
i, j, k: Integer;
Ziffer: 0..9;
c: Farbe;
Fertig, Fehler: Boolean;
Operator: (plus, minus, mal);
Hintergrund: set of Farbe;
Hintergrundfarben: array[0..Farbe-1] of Byte; { berechnet }
Heute: Datum;
Ergebnis: Messliste;
P1, P2: Person;
Matrix: array[1..10, 1..10] of Real;
  
```

Wenn ein Bezeichner in der Bezeichnerliste eines Deklarationsteils steht, gilt er innerhalb des gesamten Blocks, für den er deklariert wurde. Es kann über den ganzen Block hinweg auf diese Variable Bezug genommen werden, solange nicht in einem untergeordneten Block derselbe Bezeichner für eine andere Variable verwendet wird (Redeklaration). Eine redeklarierte Variable benutzt den Namen eines bereits existierenden Bezeichners, stellt aber ansonsten eine eigenständige Einheit dar. Der Wert der ursprünglichen Variablen wird durch eine Redeklaration nicht beeinflusst.

Variablen, die außerhalb von Prozeduren und Funktionen deklariert sind, werden als *global* bezeichnet und im Daten-Segment gespeichert. Innerhalb von Prozeduren und Funktionen deklarierte Variablen sind *lokal*, ihre Speicherung erfolgt im Stack-Segment.

Das Daten-Segment

Der letzte Schritt des Compilers besteht aus einem Verbinden des erzeugten Codes und der dazugehörigen Daten zu einem lauffähigen Programm. Bei diesem Schritt wird sämtlichen globalen Variablen des Programms sowie sämtlichen globalen Variablen aufgenommener Units ein Platz im Daten-Segment zugeordnet, das maximal 65520 Bytes umfassen kann.

Wenn Sie mehr als 65520 Bytes für globale Variablen benötigen, müssen Sie mit dynamischer Speicherverwaltung arbeiten, d.h. den Platz für größere Variablenkonstrukte über Zeiger während der Laufzeit des Programms belegen. Der Abschnitt »Zeiger und dynamische Variablen« weiter hinten in diesem Kapitel erläutert die Details.

Das Stack-Segment

Die Größe dieses Segments wird über den Compiler-Befehl `$M` festgelegt und kann zwischen 1024 und 65520 Bytes betragen; standardmäßig gibt Turbo Pascal 16384 Bytes vor.

Jede Prozedur oder Funktion belegt bei ihrer Aktivierung (d.h. ihrem Aufruf) Platz für übergebene Parameter und lokale Variablen im Stack-Segment, der beim Rücksprung zur Aufrufstelle automatisch wieder freigegeben wird.

Der Gesamtbedarf an Speicherplatz im Stack-Segment ergibt sich aus der Summe des Einzelbedarfs sämtlicher Prozeduren und Funktionen, die zu einem gegebenen Zeitpunkt gleichzeitig aktiv sein können – er darf die vorgegebene Größe des Stack-Segments nicht überschreiten. Turbo Pascal enthält aus diesem Grund den Compiler-Schalter `$S`, der standardmäßig aktiv gesetzt ist (`($S+)`): vor jedem Aufruf einer Prozedur oder Funktion wird geprüft, ob noch ausreichend Platz im Stack-Segment vorhanden ist. Mit dem Befehl `$S-` läßt sich diese automatische Überprüfung abschalten (was erst geschehen sollte, wenn ein Programm komplett fehlerbereinigt ist – Stack-Überläufe haben meist einen vollständigen Absturz des Systems zur Folge).

Absolute Variablen

In Turbo Pascal kann die Speicheradresse einer Variablen über den Zusatz **absolute** auch direkt (d.h. über die Angabe eines fixen Wertes) festgelegt werden:

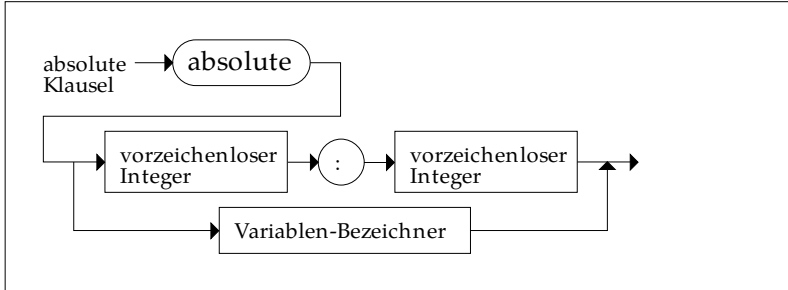


Abbildung 4.2 Syntax einer absoluten Variablendeklaration

Im Gegensatz zu allen anderen Deklarationen kann über den Zusatz **absolute** jeweils nur eine Variable vereinbart werden. Die erste Form einer Deklaration mit **absolute** gibt die Startadresse der Variablen in der Form *Segment:Offset* an:

```

var
  CrtModus:   Byte absolute $0040:$0049;
  MonoScreen: array[1..25,1..80] of
    record
      Zeichen, Attribut: Char
    end
    absolute $B000:$0000;
  
```

Die erste Konstante gibt die Segmentadresse an, die zweite den *Offset* (d.h. die Adresse innerhalb dieses Segments). Beide Konstanten müssen im Bereich von \$0000 bis \$FFFF (dezimal: 0..65535) liegen.

Die zweite Form deklariert eine Variable, die denselben Speicherplatz wie eine andere Variable belegt, und stellt praktisch eine Alternative zu varianten Teilen eines Records dar:

```

var
  Str:   string[32];
  StrLen: Byte absolute Str;
  
```

In diesem Beispiel wird festgelegt, daß die Variable *StrLen* auf derselben Adresse wie das erste Byte der Variablen *Str* gespeichert wird. Da das erste Byte eines Strings die Länge enthält, kann die momentane Länge von *Str* hier also auch über *StrLen* ermittelt werden.

Der Bezug auf Variablen

Der Bezug auf eine Variable bezeichnet:

- eine Variable;
- eine Komponente eines strukturierten Typs oder einer Variablen vom Typ *String*;
- eine dynamische Variable, auf die eine Zeiger-Variablen zeigt.

Das folgende Syntaxdiagramm verdeutlicht die Referenzmöglichkeit auf eine Variable:

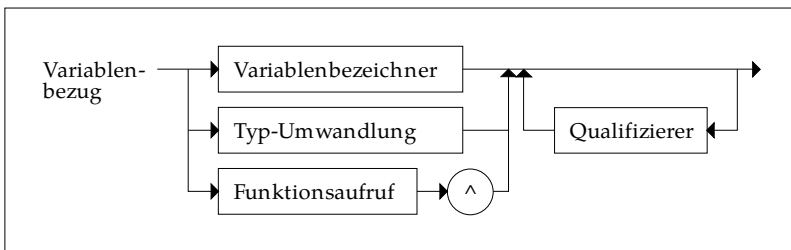


Abbildung 4.3 Syntax eines Variablenbezugs

Hinweis: Die Syntax eines Variablenbezugs ermöglicht den Aufruf von Funktionen, die als Ergebnis einen Zeiger zurückliefern. Dieses »Zeiger-Ergebnis« wird von Turbo Pascal automatisch in die tatsächliche Adresse der Variablen umgewandelt (de-referencing).

Die Festlegung eines Variablenbezugs

Der Bezug auf eine Variable geschieht über einen Variablenbezeichner, dem optional ein oder mehrere Werte zur Qualifizierung folgen können. Diese Werte (Qualifizierer) legen fest, in welchem Zusammenhang auf eine Variable Bezug genommen wird.

Ein Array-Bezeichner ohne zusätzliche Angaben bezieht sich auf das gesamte Array. Ein Beispiel:

Ergebnisse

Ein Array-Bezeichner mit einem darauffolgenden Index steht für eine bestimmte Komponente des Arrays:

Ergebnisse[Index + 1]

Ist diese Komponente ein Record oder ein Objekt, kann dem Index ein

Feldbezeichner angefügt werden. Der Variablenzugriff bezeichnet in diesem Fall ein bestimmtes Feld in einer bestimmten Komponente des Arrays.

`Ergebnisse[Index + 1].Daten`

Auf den Bezeichner eines Feldes, das einen Zeigerwert enthält, kann das Zeigersymbol `^` folgen. Dieses Symbol unterscheidet zwischen dem Feld selbst und der dynamischen Variablen, auf die sein Inhalt zeigt.

`Ergebnisse[Index + 1].Daten^`

Wenn die Variable, auf die ein Zeiger zeigt, ein Array ist, können zusätzliche »Indexe« hinzugefügt werden, die für einzelne Komponenten dieses Arrays stehen.

`Ergebnisse[Index + 1].Daten^[J]`

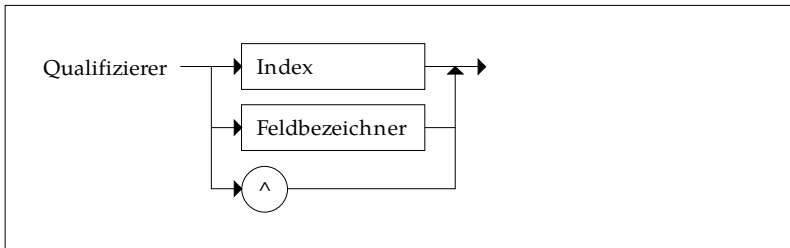


Abbildung 4.4 Syntax von Werten zur Qualifizierung von Variablenbezeichnern

Arrays, Strings und Indizierungen

Der Bezug auf eine Komponente einer Array-Variablen geschieht über die Angabe eines Index nach dem Bezeichner der Array-Variablen. Der Bezug auf ein Zeichen innerhalb einer String-Variablen geschieht auf dieselbe Weise, nämlich ebenfalls über die Angabe eines Index nach dem Bezeichner der String-Variablen.

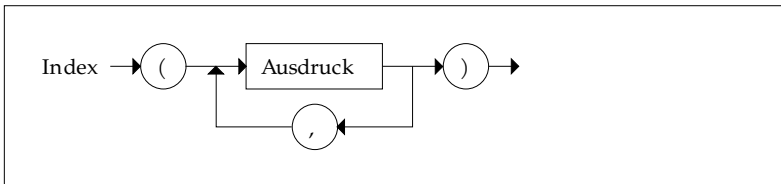


Abbildung 4.5 Syntax von Index-Ausdrücken

Die Index-Ausdrücke wählen die Komponenten aus den entsprechenden Dimensionen des Arrays aus. Die Anzahl der Index-Ausdrücke darf die der Index-Typen innerhalb der Array-Deklaration nicht überschreiten. Der Typ

jedes Ausdrucks muß zu dem des entsprechenden Index-Typen zuweisungskompatibel sein. Bei der Indizierung mehrdimensionaler Arrays können mehrere Notationsformen verwendet werden. Die folgenden beiden Variablenbezüge sind identisch:

```
Matrix[i][j]
Matrix[i,j]
```

Ein Element einer String-Variablen (d.h. ein einzelnes Zeichen) läßt sich durch einen einzigen Index-Ausdruck darstellen. Der Wert dieses Ausdrucks liegt im Bereich von 0 bis n , wobei n die deklarierte Größe des Strings darstellt. Das Ergebnis eines derartigen Bezugs hat den Typ *Char*.

Das erste Zeichen der String-Variablen (Index 0) enthält die dynamische Länge des Strings (d.h. $Length(S)$ entspricht $Ord(S[0])$). Direkte Zuweisungen an das Längenattribut sind – syntaktisch gesehen – zulässig, der Compiler nimmt keine spezielle Prüfung vor.

Indizierungen außerhalb der momentanen Länge eines Strings lesen »zufällige« Werte bzw. verändern den aktuellen dynamischen Stringinhalt nicht. Zuweisungen mit Indizierungen, die außerhalb der deklarierten Maximallänge eines Strings liegen, haben dieselbe Wirkung wie unzulässige Indizierungen eines Arrays: sie zerstören andere Variablen.

Records und Feldbezeichner

Der Bezug auf ein Feld eines Records geschieht über einen Bezug auf die jeweilige Record-Variable, deren Bezeichner von einem Punkt und dem Feldnamen gefolgt wird.

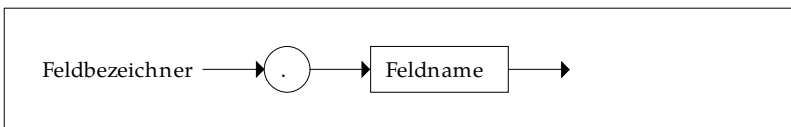


Abbildung 4.6 Syntax eines Feldbezeichners

Gültige Beispiele für Feldbezeichner sind:

```
Heute.Jahr
Messdaten[1].Zaehlung
Messdaten[1].wann.Monat
```

Anweisungen, die innerhalb eines **with**-Blocks stehen und sich auf Felder eines mit **with** adressierten Records beziehen, müssen den Record-Bezeichner selbst nicht mehr explizit aufführen.

Bezeichner von Objektkomponenten

Das Format eines Objektkomponentenbezeichners ist dasselbe wie bei einem Feldbezeichner eines Records (siehe Kapitel 3 »Typen«). Hinter der Instanz (einer Variablenreferenz) kommt ein Punkt und ein Komponentenbezeichner. Einen Komponentenbezeichner für eine Methode nennt man *Methodenbezeichner*. Auf eine Instanz eines Objekttyps kann eine **with**-Anweisung angewendet werden. In diesem Fall darf man die Instanz und den Punkt beim Zugriff auf die Komponenten des Objekttyps weglassen.

Die Instanz und den Punkt kann man auch innerhalb von Methodenblöcken weglassen. Dies entspricht der Angabe von *Self* und einem Punkt vor der Komponentenreferenz.

Zeiger und dynamische Variablen

Ein Zeiger hat entweder den Wert **nil** oder enthält die Adresse einer (dynamischen) Variablen. Im letzteren Fall *zeigt* er auf diese Variable.

Zeigt eine Zeiger-Variable auf eine dynamische Variable, wird der Bezug über das Zeigersymbol **^** (hinter dem Bezeichner der Zeiger-Variablen) notiert. Dynamische Variablen und ihre entsprechenden Zeigerwerte werden mit der Standard-Prozedur *New* erzeugt. Der Operator **@** und die Standardprozedur *Ptr* erzeugen Zeigerwerte, die in derselben Weise wie Zeiger auf dynamische Variablen behandelt werden. Der Wert **nil** steht für »not in list« und »zeigt« nirgendwohin. Ein Zugriff über einen Zeiger mit dem Wert **nil** hat einen Laufzeitfehler zur Folge. Gültige Beispiele für Bezüge auf Variablen über Zeiger sind:

```
P1^
P1^.Geschwister
Ergebnisse[1].Daten^
```

Umwandlung von Variablentypen

Der Bezug auf eine Variable eines bestimmten Typs kann in den Bezug auf eine Variable eines anderen Typs umgewandelt werden.

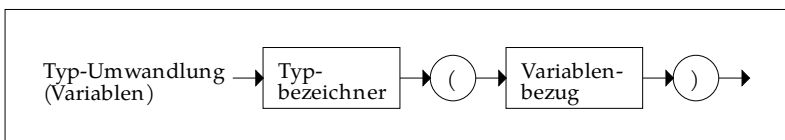


Abbildung 4.7 Syntax der Umwandlung eines Variablentyps

Wenn ein Variablenbezug umgewandelt wird, gilt dieser Bezug als Prototyp des Typen, der durch den Typ-Bezeichner festgelegt wurde. Die Größe der Variablen (also die Anzahl von Bytes, die sie im Speicher belegt) muß exakt der Größe des Typs entsprechen, für den der Typ-Bezeichner steht. Auf die Umwandlung eines Variablen-Typs kann eine beliebige Anzahl von Indizierungen folgen, soweit sie für den angegebenen Typ zulässig sind.

Einige Beispiele für die Umwandlung von Variablen-Typen:

```
type Punkt = record
    x,y: Integer;
end;
    Liste = array[1..2] of Integer;
var P : Punkt;
    L : LongInt;
    N : Integer,

begin
    P := Punkt(L);
    N := Punkt(L).x;
    LongInt(P) := LongInt(P) + $00080008;
    Liste(P) [N] := 32;
end.
```

Mit diesem Prinzip läßt sich auch eine Erweiterung der Standardfunktionen *Lo* und *Hi* von *Byte*- auf *Word*-Variablen konstruieren. Nach der Definition eines Record-Typs wie

```
type WordRec = record
    Low, High: Word;
end;
```

kann das Programm mit den folgenden Konstruktionen auf das nieder- bzw. höherwertige Wort einer *LongInt*-Variable *L_Var* zugreifen:

```
WordRec(L_Var).Low    { niederwertiges Wort }
WordRec(L_Var).High   { höherwertiges Wort }
```

Analoges gilt für das »Zerlegen« eines Zeigerwertes in seine beiden Anteile *Segment* und *Offset*:

```
program PointerPartDemo

type PtrParts = record
    Ofc, Seg: Word; { Offset auf der niedrigeren Adresse! }
end;

var p: Pointer;

begin
    p := Ptr($1234, $4597);
    WriteLn(PtrPart(p).Ofc); { $4597 }
    WriteLn(PtrPart(p).Seg); { $1234 }
end.
```

Hinweis: Turbo Pascal ermöglicht zusätzlich die Typ-Umwandlung des Wertes eines Ausdrucks. Erläuterungen dazu finden Sie in Kapitel 6 »Ausdrücke«.

Außerdem unterstützt der Compiler die Typ-Umwandlung von prozeduralen Typen. Mit den Deklarationen

type

Func = **function**(X: Integer): Integer;

var

F : Func;

P : Pointer;

N : Integer;

können beispielsweise die folgenden Anweisungen konstruiert werden:

F := Func(P);	{ Umwandlung von P in den Typ Func und Zuweisung }
@F := P;	{ direkte Zuweisung von P an die als Pointer umgewandelte Prozedur-Variable F }
Func(P) := F;	{ Zuweisung von F an die temporär umgewandelte Pointer-Variable P }
P := @F;	{ dito, nur daß hier F zuvor in den Typ Pointer gewandelt wurde }
N := F(10);	{ Aufruf einer Funktion über F und Zuweisung des Funktionsergebnisses an die Variable N }
N := Func(P)(10);	{ dito, nur daß der Aufruf hier über P erfolgt und P zuvor in den Typ Func gewandelt wurde }

Hinweis: Die Anwendung auf Prozedur-Variablen stellt den einzigen Fall dar, in dem der Adreß-Operator @ auf der linken Seite einer Zuweisung erscheinen kann.

5. Typisierte Konstanten

Typisierte Konstanten sind mit initialisierten Variablen vergleichbar, d.h. mit Variablen, deren Werte beim Aufruf ihres jeweiligen Blocks festgelegt werden. Anders als bei einer untypisierten Konstanten enthält die Deklaration sowohl den Typ als auch den Wert (siehe Abschnitt »Deklaration von Konstanten« in Kapitel 1 »Sprachelemente und Konstanten«).

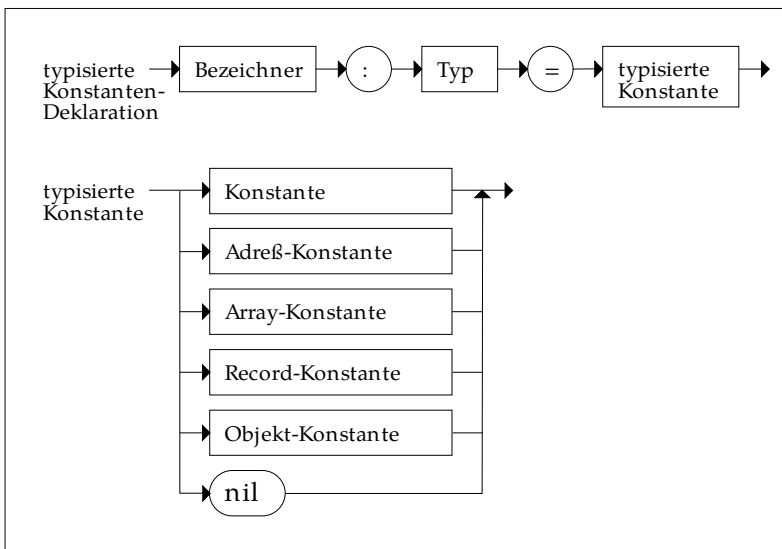


Abbildung 5.1 Syntax einer typisierten Konstantendeklaration

Typisierte Konstanten können genauso wie Variablen des entsprechenden Typs verwendet werden und dürfen auf der linken Seite einer Zuweisung erscheinen. Bitte beachten Sie, daß derartige Konstanten *einmal* initialisiert werden, nämlich beim Start des Programms. Das gilt auch für lokale Deklarationen: die Initialisierung wird also *nicht* bei jedem Aufruf der entsprechenden Prozedur oder Funktion wiederholt.

Statt mit normalen konstanten Ausdrücken können die Werte von typisierten Konstanten auch mit Hilfe von konstanten Adreß-Ausdrücken definiert

werden. Ein konstanter Adreß-Ausdruck ist ein Ausdruck, der die Adresse, das Offset oder das Segment einer globalen Variable, einer typisierten Konstante, einer Prozedur oder einer Funktion referenziert. Konstante Adreß-Ausdrücke können keine lokalen Variablen oder dynamischen Variable enthalten, weil ihre Adressen zur Compile-Zeit nicht bekannt sind.

Einfache typisierte Konstanten

Die Deklaration einfacher typisierter Konstanten enthält lediglich den Typ und einen Wert, wie die Beispiele auf der nächsten Seite zeigen:

```
const
    Maximum    : Integer = 9999;
    Faktor      : Real    = -0.1;
    CtrlC       : Char    = #3;
```

Wie oben schon angedeutet, kann der Wert einer typisierten Konstanten durch einen konstanten Adreß-Ausdruck definiert werden. Hierzu ein Beispiel:

```
var
    Buffer: array[0..1023] of Byte;

const
    BufferOfs: Word= OfS(Buffer);
    BufferSeg: Word= Seg(Buffer);
```

Da eine typisierte Konstante technisch gesehen eine Variable mit einem vorgegebenen Wert darstellt, ist sie nicht mit »echten« Konstanten austauschbar. Der Versuch, sie in der Deklaration anderer Konstanten und/oder Typen zu verwenden, erzeugt eine Fehlermeldung des Compilers:

```
const
    Min  : Integer = 0;
    Max  : Integer = 99;

type
    Vecs: array[Min..Max] of Byte; { Cannot evaluate this expression }
```

Da *Min* und *Max* hier typisierte Konstanten sind, ist ihr Wert dem Compiler unbekannt – folglich endet die Definition des Typs *Vektor* mit einer Fehlermeldung.

Strings

Die Deklaration einer typisierten String-Konstanten enthält die Maximallänge des Strings und seinen Wert:

Der Komponenten-Typ einer Array-Konstanten ist beliebig – nur Dateien sind nicht erlaubt. Gepackte String-Typen (d.h. Zeichen-Arrays) können entweder als einzelne Zeichen oder auch als Strings angegeben werden. Die beiden folgenden Deklarationen sind deshalb identisch:

const

```
Digits: array[0..9] of Char = ('0', '1', '2', '3', '4',  
                               '5', '6', '7', '8', '9');
```

const

```
Digits: array[0..9] of Char = '0123456789';
```

Bei mehrdimensionalen Array-Konstanten werden die einzelnen Elemente jeder Dimension in einem eigenen Klammerpaar angegeben, die Klammerpaare gleicher Ebenen sind durch Kommas voneinander getrennt. Eine Deklaration wie

type

```
Kubus = array[0..1, 0..1, 0..1] of Integer;
```

const

```
Wuerfel : Kubus = (((0,1), (2,3)), ((4,5), (6,7)));
```

erzeugt ein Array *Wuerfel* mit den folgenden Werten:

```
Wuerfel[0,0,0] = 0  
Wuerfel[0,0,1] = 1  
Wuerfel[0,1,0] = 2  
Wuerfel[0,1,1] = 3  
Wuerfel[1,0,0] = 4  
Wuerfel[1,0,1] = 5  
Wuerfel[1,1,0] = 6  
Wuerfel[1,1,1] = 7
```

Records

Die Deklaration eines Records als typisierte Konstante enthält Bezeichner und Wert jedes Feldes, eingeschlossen in Klammern und durch Kommas voneinander getrennt:

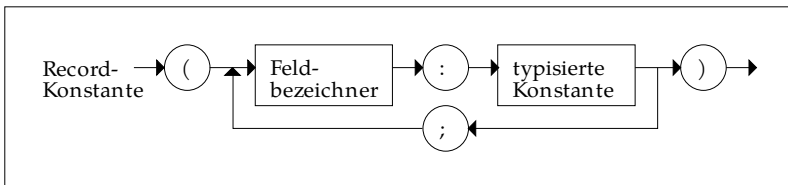


Abbildung 5.3 Syntax von Records als typisierte Konstanten

Einige Beispiele dazu:

type

```
Punkt = record
    x, y: Real;
end;
Vektor = array[0..1] of Punkt;
Monat = (Jan, Feb, Mar, Apr, Mai, Jun, Jul, Aug, Sep, Okt, Nov, Dez);
Datum = record
    Tag: 1..31; Mon: Monat; Jahr: 1900..1999;
end;
```

const

```
Ursprung: Punkt = (x: 0.0; y: 0.0);
Linie : Vektor = ((x: -3.1; y: 1.5), (x: 5.8; y: 3.0));
Heute : Datum = (Tag: 4; Monat: Okt; Jahr: 1987);
```

Die Felder müssen in derselben Reihenfolge wie bei der Definition des Record-Typs angegeben werden. Wenn ein Record Dateitypen enthält, kann das entsprechende Feld in einer Konstanten nicht deklariert werden.

Wenn der Record variante Teile enthält, ist bei einer typisierten Konstanten nur die Angabe der aktiven Varianten möglich; enthält der Record zusätzlich ein Selektorfeld, so muß dessen Wert entsprechend angegeben sein.

Konstanten mit Objekttypen

Bei der Deklaration einer Objekttypkonstanten wird die gleiche Syntax wie bei der Deklaration von Konstanten mit Record-Typen verwendet. Für die Methoden wird (und darf auch) kein Wert angegeben werden. Zu den bereits vorher angegebenen Typdeklarationen einige Beispiele mit Objekttypkonstanten:

const

```
ZeroPoint : Point = (X : 0; Y : 0);
ScreenRect : Rect =
    (A : (X : 0, Y : 0); B : (X : 80; Y : 25));
CountField : NumField = (X : 5; Y : 20; Len : 4; Name : nil;
    Value : 0; Min : -999; Max : 999);
```

Konstanten eines Objekttyps, die virtuelle Methoden enthalten, müssen *nicht* mit Hilfe eines Konstruktoraufrufs initialisiert werden. Diese Initialisierung wird automatisch vom Compiler vorgenommen.

Mengen

Die Deklaration einer Menge als typisierte Konstante enthält die Elemente der Menge, eingeschlossen in eckige Klammern und durch Kommas voneinander getrennt.

Sowohl die Angabe einer leeren Menge (0 Elemente) als auch die Angabe einer Anzahl von Elementen (kleinstes Element, zwei Punkte, größtes Element) ist möglich. Einige Beispiele dazu:

type

```
Digits = set of 0..9;
Letters = set of 'A'..'Z';
```

const

```
GeradeZiffern : Digits = [0,2,4,6,8];
Vokale        : Letters = ['A', 'E', 'I', 'O', 'U'];
HexZiffern    : set of '0'..'Z' = ['0'..'9', 'A'..'F', 'a'..'f'];
```

Zeiger

Deklarationen von Zeigern als typisierte Konstante gebrauchen typischerweise einen konstanten Adreß-Ausdruck zur Definition des Zeiger-Wertes. Hierzu einige Beispiele:

type

```
Richtung = (links, rechts, auf, ab);
StringPtr = ^String;
NodePtr = ^Node;
Node = record
  Naechster: NodePtr;
  Symbol: StringPtr;
  Wert: Richtung;
end;
```

const

```
S1: String[6] = 'Ab';
S2: String[6] = 'Auf';
S3: String[6] = 'Rechts';
S4: String[6] = 'Links';
N1: Node = (Naechster: nil; Symbol: @S1; Wert: ab);
N2: Node = (Naechster: @N1; Symbol: @S2; Wert: auf);
N3: Node = (Naechster: @N2; Symbol: @S3; Wert: rechts);
N4: Node = (Naechster: @N3; Symbol: @S4; Wert: links);
RichtungsTab: NodePtr = @N4;
```

Prozedurale Typen

Eine Konstante prozeduralen Typs muß solche Prozeduren oder Funktionen referenzieren, die zuweisungskompatibel zum Typ der Konstanten sind.

Ein Beispiel:

type

ErrorProc = **procedure**(ErrorCode: Integer);

procedure DefaultError(ErrorCode: Integer); **far**;

begin

WriteLn('Fehler ', ErrorCode, '.');

end;

const

ErrorHandler: ErrorProc = DefaultError;

6. Ausdrücke

Ausdrücke bestehen aus Operatoren und Operanden. Die meisten Operatoren von Turbo Pascal verknüpfen zwei Operanden und werden deshalb als *binär* bezeichnet. Die restlichen Operatoren arbeiten mit nur einem Operanden, daher bezeichnet man sie als *unär*. Binäre Operatoren benutzen die herkömmliche algebraische Form wie z.B. $a + b$. Ein unärer Operator steht immer unmittelbar vor seinem Operanden, wie bei $-b$. Bei umfangreicheren Ausdrücken regelt die in Tabelle 6.1 gezeigte *Rangfolge* der Operatoren die Reihenfolge der Berechnung. Es gelten drei grundlegende Gesetze:

- Ein Operand zwischen zwei Operatoren von unterschiedlichem Rang ist immer an den höherrangigen Operator gebunden.
- Ein Operand zwischen zwei gleichrangigen Operatoren ist immer an den Operator gebunden, der links von ihm steht.
- Ausdrücke in Klammern werden als ein einzelner Operand betrachtet und immer als erstes ausgewertet.

Tabelle 6.1 Rangfolge der Operatoren

Operatoren	Rangfolge	Kategorie
not, @	1 (am höchsten)	unär
*, /, div, mod, and, shl, shr	2	multiplizierend
+, -, or, xor	3	addierend
=, <, >, <=, >=, in	4	relational

Operationen von gleichem Rang werden normalerweise von links nach rechts durchgeführt – der Compiler kann allerdings zur Optimierung des Rechenweges Umstellungen vornehmen.

Syntax von Ausdrücken

Die Rangfolge der Operatoren folgt der Syntax von Ausdrücken, die sich aus Faktoren, Termen und einfachen Ausdrücken zusammensetzen.

Faktoren lassen sich durch das folgende Syntaxdiagramm darstellen.

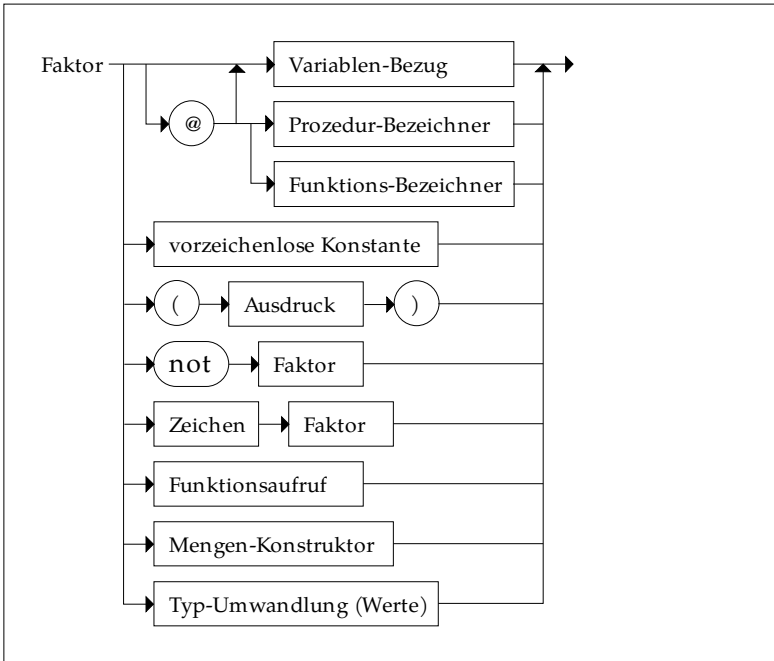


Abbildung 6.1 Syntax von Faktoren

Ein Funktionsaufruf aktiviert eine Funktion und kann deshalb mit dem von ihr zurückgelieferten Wert gleichgesetzt werden (vgl. Abschnitt »Funktionsaufrufe« in diesem Kapitel).

Eine Mengenangabe steht für den Wert eines Mengen-Typs (vgl. »Mengen-Konstrukturen«). Eine vorzeichenlose Konstante hat die in Abbildung 6.2 dargestellte Syntax.

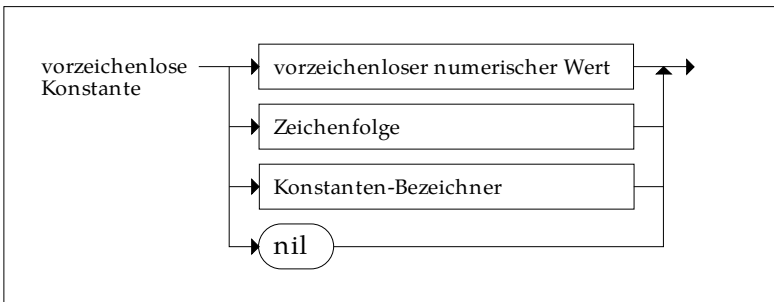


Abbildung 6.2 Syntax einer vorzeichenlosen Konstanten

Die folgenden Angaben stellen gültige Faktoren dar:

X	Variablenbezug
MyUnit.X	qualifizierter Variablenbezug
@X	Adresse einer Variablen
@@MathPtr	Adresse einer Prozedur-Variablen
15	vorzeichenlose Konstante
(X+Y+Z)	Unterausdruck
Sin(X / 2)	Funktionsaufruf
MathPtr(X)	Funktionsaufruf über Prozedur-Variable
['0'..'9', 'A'.. 'Z']	Mengenangabe
not done	Negation von Boolean
Char(Digit+48)	Umwandlung des Typs eines Wertes

Terme ordnen Faktoren Multiplikations-Operatoren zu:

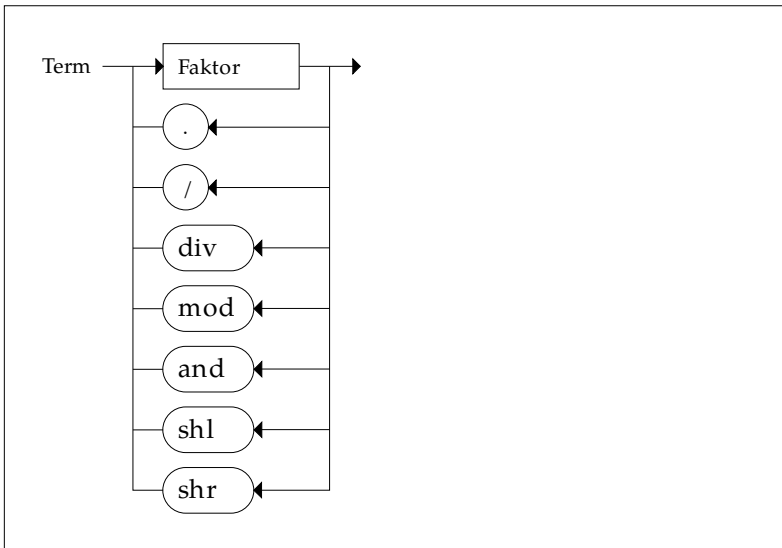


Abbildung 6.3 Syntax von Termen

Beispiele für gültige Terme sind:

X * Y
 Z / (1-Z)
 Done or Error
 (X <= Y) and (Y < Z)

Einfache Ausdrücke ordnen Termen Additionsoperatoren und Zeichen zu, wie in Abbildung 6.4 dargestellt. Einige Beispiele dazu:

X + Y
 -X

Hue1 + Hue2

I * J + 1

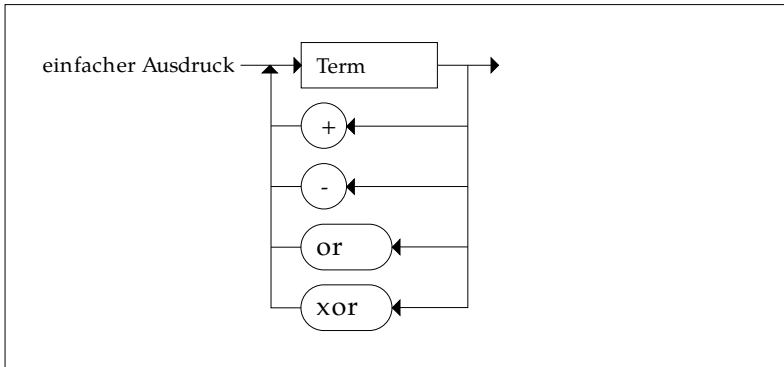


Abbildung 6.4 Syntax einfacher Ausdrücke

Ein Ausdruck ordnet einfachen Ausdrücken relationale Operatoren zu:

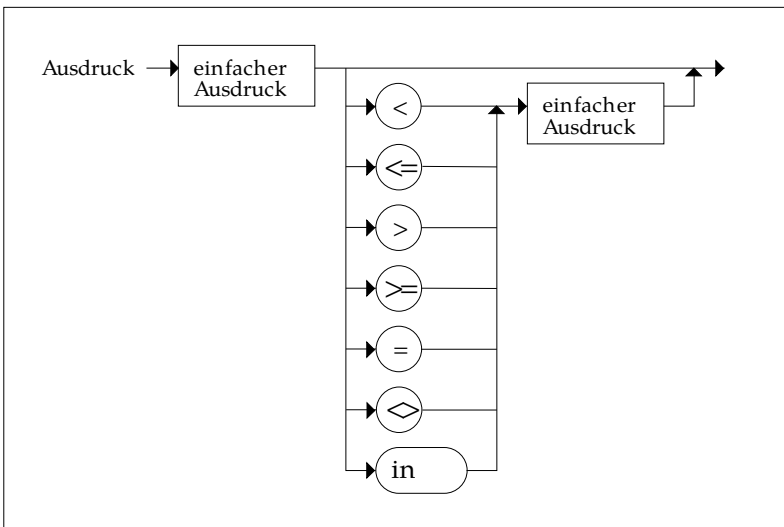


Abbildung 6.5 Syntax eines Ausdrucks

Einige Beispiele für Ausdrücke:

X = 1.5

Done <> Error

(I < J) = (J < K)

C in Hue1

Operatoren

Operatoren werden in sechs Gruppen unterschieden: arithmetische, logische, String-, Mengen- und relationale Operatoren sowie den Adreß-Operator @.

Arithmetische Operatoren

Die Tabellen 6.2 und 6.3 zeigen die Operanden- und Ergebnistypen binärer bzw. unärer arithmetischer Operationen.

Tabelle 6.2 Binäre arithmetische Operationen

Operator	Operation	Operandentyp	Ergebnistyp
+	Addition	Integer	Integer
		Real	Real/Extended
-	Subtraktion	Integer	Integer
		Real	Real/Extended
*	Multiplikation	Integer	Integer
		Real	Real/Extended
/	Division	Integer	Integer
		Real	Real/Extended
div	Integerdivision	Integer	Integer
mod	Modulo	Integer	Integer

Hinweis: Der Operator + wird auch als Stringoperator benutzt, die Operatoren +, - und * werden auch als Mengenoperatoren benutzt.

Tabelle 6.3 Unäre arithmetische Operationen

Operator	Operation	Operandentyp	Ergebnistyp
+	Identität	Integer	Integer
		Real	Real/Extended
-	Negation	Integer	Integer
		Real	Real/Extended

Operanden, die Teilbereiche ordinaler Typen darstellen, werden auf dieselbe Weise wie der »ursprüngliche« Ordinaltyp behandelt. Wenn beide Operanden eines Operators +, -, *, **div** oder **mod** einen Integer-Typ haben, hat das Ergebnis den Typ des gemeinsamen Formats (siehe Abschnitt »Der Typ Integer« in Kapitel 3 »Typen«).

Ist einer der Operanden eines Operators +, - oder * vom Typ *Real*, dann hat das Ergebnis im Modus \$N- den Typ *Real*, im Modus \$N+ den Typ *Extended*. Ist der Operand eines Identitäts- oder Negations-Operators ein Integer-Typ, dann

ist auch das Ergebnis im selben Format. Ist der Operator dagegen ein Real-Typ, dann ergibt sich – je nach Stand des Schalters \$N – ein Wert vom Typ *Real* oder *Extended*. Das Ergebnis von x/y ist immer vom Typ *Real* bzw. *Extended*, wobei der Typ der Operanden keine Rolle spielt. Wenn y den Wert 0 hat, ergibt sich ein Laufzeitfehler.

Der Wert von $i \text{ div } j$ ist der mathematische Quotient von i/j , auf einen Integerwert abgerundet. Wenn j den Wert 0 hat, ergibt sich ein Laufzeitfehler.

Der Operator **mod** liefert den Rest der Division seiner beiden Operanden zurück, also:

$$i \text{ mod } j = i - (i \text{ div } j) * j$$

Das Ergebnis einer Operation mit **mod** übernimmt das Vorzeichen seines Operanden i . Wenn j den Wert 0 hat, ergibt sich ein Laufzeitfehler.

Logische Operatoren

Tabelle 6.4 gibt die Typen der Operanden und Ergebnisse logischer Operationen wieder.

Tabelle 6.4 Logische Operationen

Operator	Operation	Operandentyp	Ergebnistyp
not	bitweise Negation	Integer	Integer
and	bitweises UND	Integer	Integer
or	bitweises ODER	Integer	Integer
xor	bitweise Antivalenz	Integer	Integer
shl	Linksschieben	Integer	Integer
shr	Rechtsschieben	Integer	Integer

Hinweis: **not** ist ein unärer Operator.

Wenn der Operand von **not** einen Integer-Typ hat, dann ist auch das Ergebnis in diesem Format. Wenn beide Operanden von **and**, **or** oder **xor** einen Integer-Typ haben, dann hat das Ergebnis das gemeinsame Format der beiden Operanden (vgl. Kapitel 3 »Typen«).

Die Operationen $i \text{ shl } j$ und $i \text{ shr } j$ verschieben den Wert von i um j Bitpositionen nach links bzw. nach rechts, entsprechen also einer Multiplikation bzw. Division mit 2^j . Der Typ des Ergebnisses ist der Typ von i .

Boolesche Operatoren

Tabelle 6.5 gibt die Typen der Operanden und Ergebnisse boolescher Operationen wieder.

Tabelle 6.5 Boolesche Operationen

Operator	Operation	Operandentyp	Ergebnistyp
not	logische Negation	Boolean	Boolean
and	logisches UND	Boolean	Boolean
or	logisches ODER	Boolean	Boolean
xor	logische Antivalenz	Boolean	Boolean

Hinweis: Der Operator **not** ist auch hier unär.

Bei Operanden des Typs *Boolean* bestimmt die normale boolesche Logik das Ergebnis dieser Operationen. Beispielsweise ergibt *a* **and** *b* nur dann *True* (wahr), wenn *a* und *b* wahr sind.

Turbo Pascal unterstützt zwei Arten der Code-Erzeugung für die Auswertung zusammengesetzter Ausdrücke mit den Operatoren **and** und **or**: vollständige Berechnung und das *Kurzschlußverfahren*:

■ Bei einer vollständigen Berechnung werden sämtliche Teile des Ausdrucks berechnet – selbst dann, wenn das Ergebnis bereits feststeht, also beispielsweise der erste Teil einer **and**-Operation den Wert *False* ergibt. Diese Art der Auswertung ist z.B. erforderlich, wenn Teile des Ausdrucks aus Funktionsaufrufen bestehen, über die weitere Operationen ausgeführt und/oder Werte verändert werden (Seiteneffekt).

■ Das Kurzschlußverfahren garantiert eine strikte Auswertung von links nach rechts und bricht ab, sobald das Ergebnis des Gesamtausdrucks feststeht – beispielsweise dann, wenn der erste Teil einer **or**-Operation den Wert *True* ergibt. Die Art der Auswertung erhöht nicht nur die Arbeitsgeschwindigkeit des Programms und erzeugt normalerweise kompakteren Code, sondern ermöglicht auch Konstrukte, die bei vollständiger Berechnung nicht legal wären.

Zwei Beispiele dazu:

```
while (I <= Length(S)) and (S[I] <> ' ') do Inc(I);
while(P <> nil) and (P^.Wert <> 5) do P := P^.Next;
```

Hier wird in beiden Fällen beim Kurzschlußverfahren die zweite Prüfung nicht durchgeführt, wenn die erste den Wert *False* ergibt. Bei einer vollständigen Berechnung ergäben sich Zugriffe auf nicht definierte Adressen als Teil der Prüfung selbst – die Bedingungen müßten deshalb folgendermaßen formuliert werden:

```
while (I <= Length(S)) do
  if S[I] <> ' ') then Inc(I);
while(P <> nil) do
  if (P^.Wert <> 5) then P := P^.Next;
```

Welche Art von Code der Compiler zur Auswertung zusammengesetzter Bedingungen generiert, wird über den Schalter *\$B* kontrolliert, der standardmäßig auf *\$B-* (Kurzschlußverfahren) gesetzt ist. Nur wenn dieser Schalter (mit *\$B+* oder via *Options/Compiler*) aktiviert ist, werden zusammengesetzte logische Ausdrücke vollständig berechnet.

Da Standard-Pascal nicht definiert, in welcher Weise logische Ausdrücke ausgewertet werden, sind Programme, die sich auf eines der beiden Verfahren verlassen, nicht vollständig portabel. Die Formulierung zusätzlicher Bedingungen (wie oben gezeigt), stellt den sichereren Weg dar, der aber aufwendiger und bei weitem nicht so übersichtlich ist.

String-Operatoren

Tabelle 6.6 gibt die Typen der Operanden und Ergebnisse von String-Operationen wieder.

Tabelle 6.6 String-Operationen

Operator	Operation	Operandentyp	Ergebnistyp
+	Verbindung	String, Char oder packed String	String

In Turbo Pascal kann der Operator + verwendet werden, um zwei String-Operanden miteinander zu verbinden. Das Ergebnis einer Operation wie $s + t$ ist die Verbindung von s und t , wobei s und t entweder ein String-Typ, ein Char-Typ oder ein gepackter String-Typ sein müssen. Das Resultat ist mit jedem String-Typ kompatibel (allerdings nicht mit den Typen *Char* und **packed String**). Wenn das Ergebnis die Länge von 255 Zeichen überschreitet, wird es nach dem 255. Zeichen abgeschnitten.

Mengen-Operatoren

Tabelle 6.7 gibt die Operatoren-Typen für Operationen mit Mengen wieder.

Tabelle 6.7 Mengen-Operationen

Operator	Operation	Operandentyp
+	Vereinigung	kompatible Mengen
-	Differenz	kompatible Mengen
*	Durchschnitt	kompatible Mengen

Die Resultate der Operationen ergeben sich aus Gesetzen der Mengenlehre:

- ein Ordinalwert c ist nur dann in $A+B$ enthalten , wenn er in A oder B enthalten ist.
- ein Ordinalwert c ist nur dann in $A-B$ enthalten, wenn er in A , aber nicht in B enthalten ist.

■ ein Ordinalwert c ist nur dann in $A*B$ enthalten, wenn er sowohl in A als auch in B enthalten ist.

Wenn a als Element des Ergebnisses einer Mengen-Operation der kleinste Ordinalwert innerhalb dieser Menge ist und b der größte, dann ist **set of $a..b$** der Typ des Ergebnisses.

Relationale Operatoren

Tabelle 6.8 gibt die Operandentypen und die Ergebnisse relationaler Operationen wieder.

Tabelle 6.8 Relationale Operationen

Operator	Operation	Operandentyp	Ergebnistyp
=	gleich	kompatible einfache, Zeiger-, Mengen-String- oder gepackte String-Typen	Boolean
<>	ungleich	kompatible einfache, Zeiger-, Mengen-String- oder gepackte String-Typen	Boolean
<	kleiner als	kompatible einfache, Zeiger-, Mengen-String- oder gepackte String-Typen	Boolean
	größer als	kompatible einfache, Zeiger-, Mengen-String- oder gepackte String-Typen	Boolean
<=	kleiner/gleich	kompatible einfache, Zeiger-, Mengen-String- oder gepackte String-Typen	Boolean
>=	größer/gleich	kompatible einfache, Zeiger-, Mengen-String- oder gepackte String-Typen	Boolean
<=	Untermenge von	kompatible Mengen-Typen	Boolean
>=	Obermenge von	kompatible Mengen-Typen	Boolean
in	Element von	linker Operand: beliebiger Ordinaltyp t rechter Operand: set of Typ t	Boolean

Vergleiche einfacher Typen

Wenn die Operanden der Operatoren $=$, $<>$, $<$, $>$, $<=$ oder $>=$ einfache Typen haben, müssen diese Typen jeweils kompatibel sein. Wenn allerdings einer der Operanden vom Real-Typ ist, kann der andere den Typ *Integer* haben.

Vergleiche von Strings

Die relationalen Operatoren $=$, $<>$, $<$, $>$, $<=$ und $>=$ vergleichen Strings nach der durch die Anordnung des IBM-Zeichensatzes vorgegebenen Reihenfolge. Es können alle Stringwerte miteinander verglichen werden, weil bekanntlich alle Stringwerte zueinander kompatibel sind.

Ein Char-Typ ist zu einem String-Typ kompatibel und wird bei Vergleichen als String mit der Länge 1 bewertet. Beim Vergleich eines gepackten String-Typs

mit n Komponenten eines anderen String-Typs wird der gepackte String als Wert eines String-Typs mit der Länge n bewertet.

Vergleiche von gepackten Strings

Die relationalen Operatoren $=$, $<>$, $<$, $>$, $<=$ und $>=$ können ebenfalls dazu verwendet werden, um zwei Werte von gepackten String-Typen miteinander zu vergleichen, wenn beide die gleiche Anzahl an Komponenten haben. In diesem Fall entspricht die Operation dem Vergleich zweier ungepackter Strings mit derselben Anzahl von Komponenten.

Vergleiche von Zeigern

Die Operatoren $=$ und $<>$ können benutzt werden, um kompatible Zeiger miteinander zu vergleichen. Zwei Zeiger sind nur dann gleich, wenn sie auf dasselbe Objekt zeigen.

Hinweis: Turbo Pascal vergleicht bei Zeigern lediglich die Einzelwerte von *Segment* und *Offset* miteinander. Aufgrund der Mehrdeutigkeit von Adressen bei Prozessoren der Reihe 80x86 können deshalb zwei logisch »verschiedene« Zeiger auf ein und dieselbe physikalische Adresse zeigen – Wertepaare wie \$0040:\$0049 und \$0000:\$0449 sind technisch gesehen identisch. Die Prozeduren *New* und *GetMem* liefern Zeiger immer in normalisierter Form zurück, d.h. mit einem *Offset*-Anteil im Bereich von \$0000 bis \$000F – hier haben Vergleiche immer das »richtige« Ergebnis. In allen anderen Fällen (z.B. bei der Wertzuordnung über die Funktion *Ptr*) muß explizit normalisiert bzw. auf möglicherweise falsche Ergebnisse von Vergleichen geachtet werden.

Vergleiche von Mengen

Wenn A und B Mengen sind, erzeugt ihr Vergleich folgende Resultate:

- $A = B$ ist nur dann wahr, wenn A und B exakt dieselben Elemente enthalten. In jedem anderen Fall ist $A \Diamond B$.
- $A \leq B$ ist nur dann wahr, wenn alle Elemente von A auch Elemente von B sind.
- $A \geq B$ ist nur dann wahr, wenn alle Elemente von B auch Elemente von A sind.

Überprüfung der Mengenzugehörigkeit

Der Operator **in** liefert den Wert *True* zurück, wenn der Wert des linken Operanden ein Ordinal-Typ und ein Element des rechten Operanden (d.h. der Menge) ist; andernfalls wird der Wert *False* zurückgeliefert.

Der Adreß-Operator @

Über den Adreß-Operator **@** kann die Adresse einer Variablen bestimmt und einem Zeiger zugeordnet werden. Tabelle 6.9 zeigt mögliche Operanden und Ergebnistypen.

Tabelle 6.9 Zeiger-Operationen

Operator	Operation	Operandentyp	Ergebnistyp
@	Adreß-Ermittlung	Variablenbezug, Funktions- oder Prozedur-Bezeichner	<i>Pointer</i> (derselbe Variablentyp wie nil)

@ ist ein unärer Operator. Als Operanden verwendet er einen Variablen-Bezug bzw. einen Prozedur- oder Funktionsbezeichner und liefert einen Zeiger auf diesen Operanden zurück (d.h. eine Adresse). Der Typ dieses Wertes entspricht dem des Typs von **nil** (»untypisierter Zeigerwert«) und kann deshalb jeder beliebigen Zeiger-Variablen zugewiesen werden.

Hinweis: Da eine prozedurale Variable zum einen eine Adresse enthält, zum anderen aber selbst eine Adresse hat, wurde die Syntax von @ entsprechend erweitert: Wenn *F* eine prozedurale Variable ist, dann liefert @*F* die Adresse der Routine, die *F* zugeordnet ist; @@*F* liefert die Adresse, ab der die Variable *F* selbst im Speicher steht (vgl. Kapitel 3 »Typen« und 8 »Prozeduren und Funktionen«).

@ mit einer Variablen

Gegeben seien folgende Deklarationen:

```

type
  ZweiChar = packed array[0..1] of Char;
var
  Int : Integer;
  ZweiCharPtr : ^ZweiChar;

```

Die folgende Anweisung weist nun dem Zeiger *ZweiCharPtr* die Adresse der Variablen *Int* zu:

```
ZweiCharPtr := @Int;
```

Da *ZweiCharPtr* als Zeiger auf ein Char-Array definiert wurde, läßt sich der Wert von *Int* nun ebenfalls als **packed array**[0..1] **of** *Char* interpretieren: *ZweiCharPtr*^[0] ist ein Bezug auf das niederwertige, *ZweiCharPtr*^[1] ein Bezug auf das höherwertige Byte von *Int*.

@ mit einem Wert-Parameter

Wenn @ auf einen formalen Wert-Parameter angewendet wird, ergibt sich ein Zeiger auf die Adresse innerhalb des Stacks, die den aktuellen Wert enthält, also

```

procedure WertZeiger (Parm1: Integer);
var
  PPtr: ^Integer;
begin
  PPtr := @Parm1;
.

```

```
end;
```

Diese Adreßermittlung ist nicht mit einem Variablen-Parameter vergleichbar – da die Routine nur eine Kopie des jeweils aktuellen Parameterwertes übergeben bekommt, bezieht sich $PPtr^{\wedge}$ nach der gezeigten Zuweisung nicht auf eine »übergebene Variable«, sondern auf eine Kopie ihres Wertes, die sich auf dem Stack befindet. Eine Zuordnung wie $PPtr^{\wedge} := 12$; verändert also nur den übergebenen Wert – nicht etwa eine Variable, mit der die Prozedur aufgerufen wurde.

@ mit einem Variablen-Parameter

Wird @ auf einen formalen Variablen-Parameter angewendet, ergibt sich ein Zeiger auf den aktuellen Parameter, dessen Adresse vom Stack gelesen wird, also:

```
prozedur VarZeiger (var Parm1: Integer);
var
  PPtr: ^Integer;
begin
  PPtr := @Parm1;
  .
  .
end;
```

In diesem Fall bezieht sich $PPtr^{\wedge}$ direkt auf die übergebene Variable, die durch eine Zuordnung wie $PPtr^{\wedge} := 12$; entsprechend verändert wird.

@ mit Prozeduren und Funktionen

Die Anwendung des Operators @ auf den Bezeichner einer Prozedur oder Funktion liefert die Startadresse der entsprechenden Routine zurück. Turbo Pascal enthält keine weiteren Mechanismen zur Verwendung dieses Wertes, der sich – im Gegensatz zu Prozedur-Variablen – jeder Kontrolle des Compilers entzieht. Der einzige Zweck der Anwendung von @ auf Routinen besteht in der Übergabe von Adressen an Maschinensprache-Unterprogramme.

@ mit Methoden

Der Operator @ kann auf qualifizierte Methoden-Bezeichner angewandt werden, wenn der Einsprungspunkt der Methode gebraucht wird. Ein Beispiel:

```
begin
  SetIntVec($5, @Screen.Hardcopy);
end;
```

Funktionsaufrufe

Ein Funktionsaufruf aktiviert die Funktion, die durch den angegebenen Bezeichner spezifiziert ist. (Jeder Bezeichner, der zur Repräsentation einer Funktion deklariert wird, ist ein Funktions-Bezeichner).

besteht aus Ausdrücken, die von eckigen Klammern umschlossen sind. Jeder dieser Ausdrücke repräsentiert ein Element der Menge.

Der Ausdruck `[]` steht für die leere Menge, die zu jedem Mengen-Typ zuweisungskompatibel ist. Eine Bereichsangabe wie `x..y` repräsentiert sämtliche Elemente einer Menge, deren ordinale Werte größer/gleich dem ordinalen Wert von `x` und kleiner/gleich dem ordinalen Wert von `y` sind. Wenn `x` größer ist als `y`, dann steht `x..y` für *kein* Element der Menge, die Angabe `[x..y]` ist in diesem Fall mit `[]` identisch.

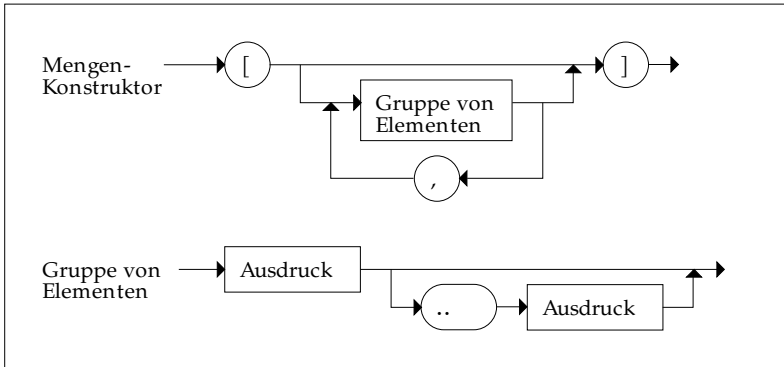


Abbildung 6.7 Syntax von Mengenkonstruktoren

Jeder Wert jedes Ausdrucks einer Gruppe von Elementen in einer Mengenangabe muß denselben Ordinal-Typ haben. Beispiele für Mengenangaben sind:

```
[rot, c, gruen]
['A'..'Z', 'a'..'z', Chr(Digit+48)]
[1, 5, 10..K mod 12, 13]
```

Umwandlung von Wert-Typen

Über eine Wert-Umwandlung (value-type cast) kann der Typ eines Ausdrucks in einen anderen Typ umgewandelt werden.

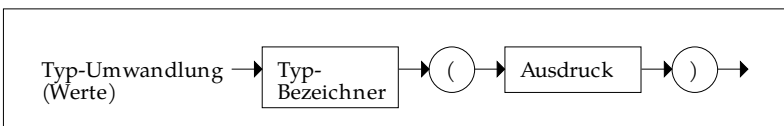


Abbildung 6.8 Syntax der Wertumwandlung

Das Argument des Ausdrucks muß entweder ein Ordinal- oder ein Zeiger-Typ sein. Das Ergebnis hat den spezifizierten Typ, sein Ordinalwert ergibt sich aus

der Umwandlung des Ausdrucks. Diese Umwandlung kann ein Abschneiden oder auch eine Erweiterung des ursprünglichen Wertes mit sich bringen, wenn die Größe des spezifizierten Typs ungleich der des Ausdrucks ist. Bei Erweiterungen von Werten bleibt das Vorzeichen erhalten.

Die Syntax zur Umwandlung eines Wert-Typs ist zu der Umwandlung eines Variablen-Typs nahezu äquivalent (siehe Kapitel 4 »Variablen«). Allerdings arbeitet diese Umwandlung mit Werten, nicht mit Variablen – deshalb kann sie nicht bei der Bezugnahme auf Variablen benutzt werden (woraus folgt, daß auf die Typ-Umwandlung eines Wertes keine Qualifizierer folgen können). Diese Art der Umwandlung darf nicht auf der linken Seite einer Zuweisung erscheinen. Gültige Beispiele für Umwandlungen von Wert-Typen sind:

```
Integer('A')
Char(48)
Boolean(0)
Farbe(2)
Longint(@Buffer)
BytePtr(Ptr($40,$49))
Boolean(2){ erlaubt, aber undefiniert }
Byte(317){ erlaubt, Vorsicht! }
```

Prozedur-Variablen in Ausdrücken

Wenn eine Prozedur-Variable in einem Ausdruck oder einer Anweisung erscheint, wird sie normalerweise mit einem Aufruf der entsprechenden Routine gleichgesetzt. Von dieser Regel muß es eine Ausnahme geben: Wenn der Compiler auf der *linken* Seite einer Zuweisung eine Prozedur-Variable erkennt, behandelt er die rechte Seite nicht als Aufruf, sondern als Wert – anders wäre die Zuweisung einer Routine an eine Variable oder die Zuweisung zwischen Prozedur-Variablen auch nicht möglich.

Ein Beispiel dazu:

```
type
  IntFunc = function: Integer;

var
  F: IntFunc;
  N: Integer;

function ReadInt: Integer; far
var
  I: Integer;
begin
  ReadLn(I);
  ReadInt := I;
end;
```

begin

```
F := ReadInt;  { Zuweisung an die Prozedur-Variable }  
N := ReadInt;  { Aufruf der Funktion ReadInt }  
N := F;        { dito }
```

end.

Die erste Zuweisung des Hauptprogramms enthält auf der linken Seite eine Prozedur-Variable – folglich codiert der Compiler hier keinen Aufruf der Funktion *ReadInt*, sondern weist diese Funktion der Variablen *F* zu (technisch: läßt die Adresse von *ReadInt* in *F* speichern).

Während sich die Frage »Aufruf oder Funktionsadresse« bei Zuweisungen immer eindeutig beantworten läßt, stellen Vergleiche den Compiler vor ein Kompatibilitätsproblem. In einer Anweisung wie

```
if F = ReadInt then Writeln('Ist gleich.');
```

ließe sich zwar ebenfalls (über den »Typ der linken Seite«) festlegen, daß hier die momentan in *F* gespeicherte Adresse mit der von *ReadInt* verglichen werden soll – das aber läuft der Sprachdefinition von Pascal zuwider, in der festgelegt ist, daß ein Funktionsbezeichner in einem *Ausdruck* gleichbedeutend mit dem Aufruf dieser Funktion ist. Turbo Pascal hält sich an diesen Standard: Der Compiler würde also im gezeigten Beispiel einen Aufruf von *ReadInt* codieren und dann versuchen, das Ergebnis dieser Funktion mit der Variablen *F* zu vergleichen (was mit einer Beschwerde über unpassende Typen endet).

Für den *Vergleich* einer Prozedur-Variablen mit der Adresse einer Routine wird deshalb eine Sonderform benutzt:

```
if @F = @ReadInt then Writeln('F zeigt auf die Funktion ReadInt');
```

Der Adreß-Operator @ steht hier für den *Inhalt* der Prozedur-Variablen *F* und macht die Absicht des Programmiers eindeutig. Technisch gesehen, steht er für eine Umwandlung in eine untypisierte Variable des Typs *Pointer* – und die läßt sich mit der (ebenfalls untypisierten) Adresse einer Funktion ohne Kompatibilitätsprobleme vergleichen.

Als Konsequenz dieser Festlegung wird die *Adresse* einer Prozedur-Variablen über die doppelte Anwendung des Operators @ ermittelt: Wenn *F* als Prozedur-Variable deklariert ist, dann liefert @*F* den in dieser Variablen gespeicherten Wert (d.h. eine Routinen-Adresse in Form eines untypisierten Zeigers); @@*F* liefert die Adresse, ab der die Variable *F* selbst im Speicher steht.

7. Anweisungen

Der englische Begriff *Statement* steht für alle Konstrukte, die eine durch den Computer ausführbare Aktion vereinbaren. Wir verwenden in diesem Handbuch den Terminus »Anweisung« als Oberbegriff für Anweisungen (wie **begin**, **end** oder **for**) und *Befehle* (wie **goto**, Zuweisungen, Prozeduraufrufe usw.).

Jeder Anweisung (d.h. jeder Vereinbarung einer ausführbaren Aktion) kann ein *Label* vorangestellt werden, auf das wiederum ein Bezug mit **goto** möglich ist: ein **goto** zu diesem Label bewirkt einen direkten Sprung zu dieser Anweisung und ihre Ausführung.

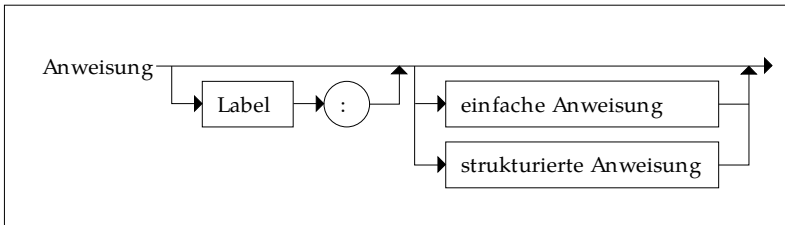


Abbildung 7.1 Syntax der *goto*-Anweisung

Wie bereits in Kapitel 1 »Sprachelemente und Konstanten« definiert, ist ein Label entweder eine Ziffernfolge im Bereich von 0 bis 9999 oder ein Bezeichner.

Anweisungen werden nach den Kriterien *einfach* und *strukturiert* unterschieden:

Einfache Anweisungen

bestehen aus Zuweisungen, Prozeduraufrufen oder Sprüngen, d.h. enthalten in sich keine weiteren Anweisungsebenen.

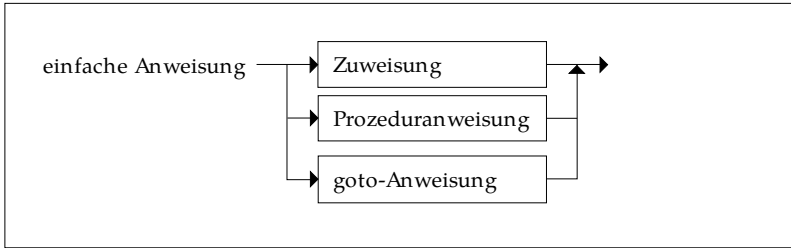


Abbildung 7.2 Syntax einer einfachen Anweisung

Zuweisungen

Zuweisungen ersetzen den momentanen Wert einer Variablen mit einem neuen Wert, der über einen Ausdruck angegeben wird. Dieser Ausdruck kann Bezeichner von Funktionen enthalten, die dadurch aktiviert werden und entsprechende Werte zurückliefern.

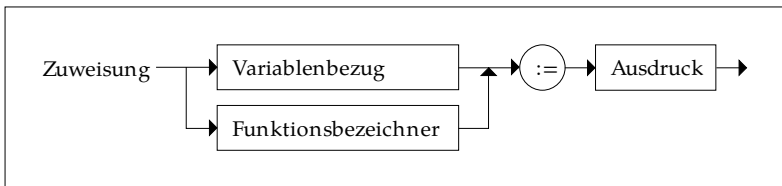


Abbildung 7.3 Syntax einer Zuweisung

Der Ausdruck muß zum Typ der Variablen oder zum Ergebnistyp der Funktion zuweisungskompatibel sein (vgl. Abschnitt »Kompatibilität von Typen« in Kapitel 4 »Variablen«). Gültige Zuweisungen sind:

```

X := Y + Z;
Done := (I>=1) and (I<100);
Farben1 := [blau, Succ(Farbe)];
I := Sqr(J) - I*K;
  
```

Die Regeln für die Zuweisungskompatibilität ermöglichen, daß der Instanz eines Objekttyps die Instanz eines ihrer Nachkommenstypen zugewiesen wird. Eine derartige Zuweisung stellt eine *Projektion* des Nachkommens auf den Raum dar, der von seinen Vorfahren umspannt wird. Wenn zum Beispiel eine Instanz *F* des Typs *Field* und eine Instanz *Z* des Typs *ZipField* vorliegt, dann werden mit der Zuweisung *F* := *Z* nur die Felder *X*, *Y*, *Len* und *Name* kopiert.

Die Zuweisung einer Instanz eines Objekttyps ist *nicht* mit einer Initialisierung der Instanz verbunden. Auf das vorherige Beispiel bezogen, bedeutet die Zuweisung *F* := *Z* nicht, daß der Konstruktoraufwurf für *F* weggelassen werden kann.

meter *Self* der Methode, die den Aufruf enthält. Eine vollständige Aktivierung verwendet nie den Aufrufmechanismus der virtuellen Methoden. Der Aufruf erfolgt immer statisch, es wird immer die angegebene Methode aufgerufen.

Eine Aktivierung mit vollständigen Bezeichnern wird im allgemeinen in einer Ersatzmethode verwendet, um die ersetzte Methode aufrufen zu können. Bezogen auf die bereits früher deklarierten Typen folgen ein paar Beispiele mit vollständigen Bezeichnern:

```
constructor NumField.Init(FX, FY, FLen : Integer;
    FName : String; FMin, FMax : Longint);
begin
    Field.Init(FX, FY, FLen, FName);
    Wert := 0;
    Min := FMin;
    Max := FMax;
end;

function ZipField.PutStr(S : String) : Boolean;
begin
    PutStr := (Length(S) = 5) and NumField.PutStr(S);
end;
```

Wie Sie sehen, kann man mit einem vollständigen Aufruf in einer Ersatzmethode den Programmcode der Methode, die man ersetzt, wiederverwenden. Die Methode des Vorfahren kann also auch in der Methode des Nachkommen aufgerufen werden, auch wenn letztere die Methode des Vorfahren ersetzt.

Die Anweisung *goto*

führt einen Sprung zum angegebenen Label aus: Das Programm wird an dem Punkt fortgesetzt, der unmittelbar auf das Label folgt. Die Syntax von **goto** ist:

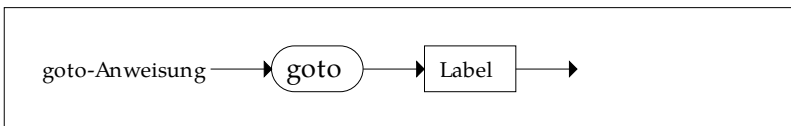


Abbildung 7.5 Syntax der *goto*-Anweisung

Bei der Verwendung von **goto** müssen die folgenden Regeln beachtet werden:

- Das Label, auf das **goto** Bezug nimmt, muß im selben Block stehen wie die **goto**-Anweisung selber. Anders gesagt: es ist nicht möglich, mit **goto** beliebig zwischen Prozeduren und/oder Funktionen hin- und herzuspringen.
- Der Bezug auf einen strukturierten Anweisungsblock von einem Programmteil außerhalb dieses Blocks (d.h. ein Sprung auf eine tiefere Verschachtelungs-

ebene) kann unvorhersehbare Folgen haben. Der Compiler erkennt derartige Fehler nicht!

Strukturierte Anweisungen

bestehen aus mehreren ineinander verschachtelten Ebenen, die ihrerseits Anweisungen enthalten. Sie werden entweder in der Reihenfolge ihres Erscheinens (Verbund- und **with**-Anweisungen), bedingt (konditionale Anweisung) oder wiederholt (Wiederholungsanweisungen bzw. Schleifen) ausgeführt.

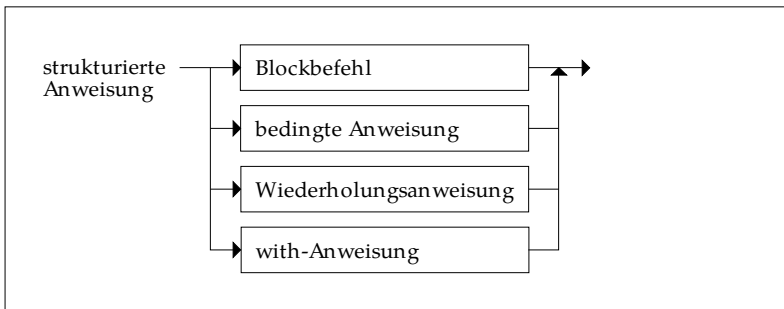


Abbildung 7.6 Syntax strukturierter Anweisungen

Verbundanweisungen

legen fest, daß diedarin enthaltenen Komponenten in der Reihenfolge ausgeführt werden, in der sie im Quelltext erscheinen. Alle im Verbund enthaltenen Anweisungen werden als ein einziger Block behandelt und genügen so den Forderungen an Stellen, wo die Syntax von Pascal nur eine einzelne Anweisung zuläßt. Anfang und Ende eines Verbundes werden durch **begin** und **end** gekennzeichnet, die Komponenten sind durch Semikolons voneinander getrennt.

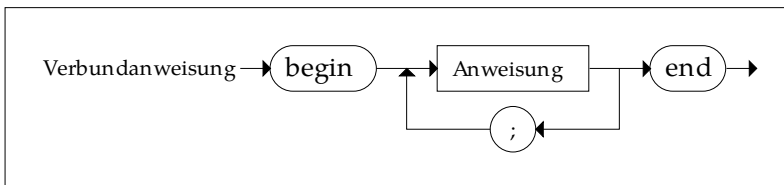


Abbildung 7.7 Syntax einer Verbundanweisung

Ein Beispiel dazu:

```
var
  X, Y, Temp: Integer;
begin
  { ... }
  if X > Y then Writeln('X größer Y.') { einfache Anweisung }
  else
    begin { Verbundanweisung }
      Temp := X; X := Y; Y := Temp;
      Writeln('X und Y vertauscht. ');
    end;
end.
```

Auch die **asm**-Anweisung ist eine Verbundanweisung. Details hierzu finden Sie im Abschnitt »Der integrierte Assembler« in Kapitel 15 »Der integrierte Assembler und Einbinden von Assembler-Code«.

Bedingte Anweisungen

bieten eine oder mehrere Optionen und wählen eine (gegebenenfalls auch keine) ihrer Komponenten zur Ausführung aus.

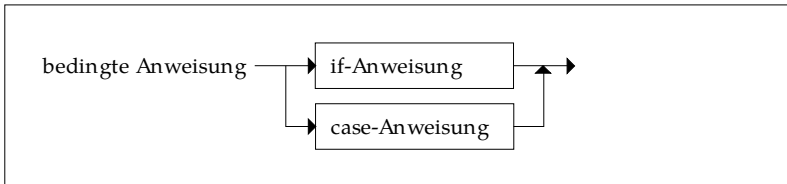


Abbildung 7.8 Syntax einer bedingten Anweisung

Die Anweisung *if*

läßt sich durch das folgende Syntaxdiagramm darstellen:

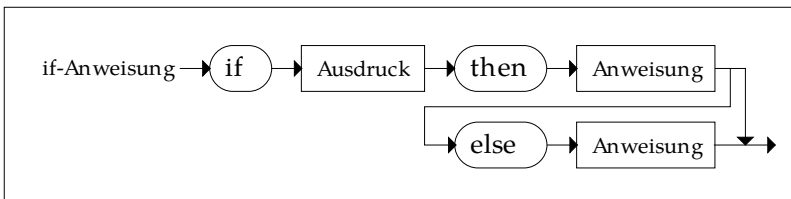


Abbildung 7.9 Syntax der if-Anweisung

Das Ergebnis eines solchen Ausdrucks muß den Standardtyp *Boolean* haben. Ergibt sich der Wert *True*, wird der auf **then** folgende Teil ausgeführt, ansonsten der auf **else** folgende Teil.

else ist optional: wenn dieser Zweig nicht vorhanden ist, bleibt die **if**-Anweisung »wirkungslos«, d.h. führt überhaupt nichts aus.

Einige Beispiele dazu:

```
if X < 1.5 then
```

```
  Z := X + Y
```

```
else
```

```
  Z := 1.5;
```

```
if P1 <> nil then
```

```
  Px := P1^.Root;
```

Mehrere aufeinanderfolgende Verschachtelungsebenen mit **if..then..else** sind syntaktisch doppeldeutig. Im folgenden Beispiel kann nicht eindeutig bestimmt werden, auf welches **if** sich das letzte **else** bezieht:

```
if e1 then if e2 then s1 else s2
```

Per Definition ist deshalb festgelegt, daß sich **else** auf das jeweils letzte **if** bezieht, zu dem noch kein **else**-Zweig angegeben wurde. Die zuvor gezeigte Folge wird also folgendermaßen interpretiert:

```
if e1 then
```

```
begin
```

```
  if e2 then
```

```
    s1
```

```
  else
```

```
    s2
```

```
end
```

Die Anweisung case

besteht aus einem Ausdruck (dem Selektor) und einer (beliebig langen) Liste von *Zweigen*. Jedem dieser Zweige gehen eine oder mehrere Konstanten (die **case**-Konstanten) oder das reservierte Wort **else** voraus. Der Selektor muß einen Ordinal-Typ haben, dessen obere und untere Grenzen im Bereich von -32768..65535 liegen – die Typen *LongInt* und *String* sind folglich als Selektor nicht erlaubt. Konstanten dürfen nicht mehrfach deklariert sein und müssen ebenfalls einen Ordinal-Typ haben, der darüber hinaus mit dem Typ des Selektors kompatibel ist.

Ein Zweig, dem eine **case**-Konstante vorausgeht, wird ausgeführt, wenn der Wert der Konstanten gleich dem des Selektors ist. Dasselbe gilt, wenn ein **case**-Bereich den Wert des Selektors enthält. Fällt der Wert des Selektors weder mit einer **case**-Konstanten noch einem **case**-Bereich zusammen, dann wird der auf das reservierte Wort **else** folgende Zweig ausgeführt. Wenn kein **else**-Zweig

definiert ist, setzt Turbo Pascal die Programmausführung mit der nächsten Anweisung nach dem **case**-Block fort.

Auf jede **case**-Konstante kann eine einfache oder eine Verbundanweisung folgen. Das reservierte Wort **else** gilt innerhalb eines **case**-Blocks als Blockbeginn: zwischen **else** und dem abschließenden **end** ist deshalb eine beliebige Anzahl von Anweisungen möglich.

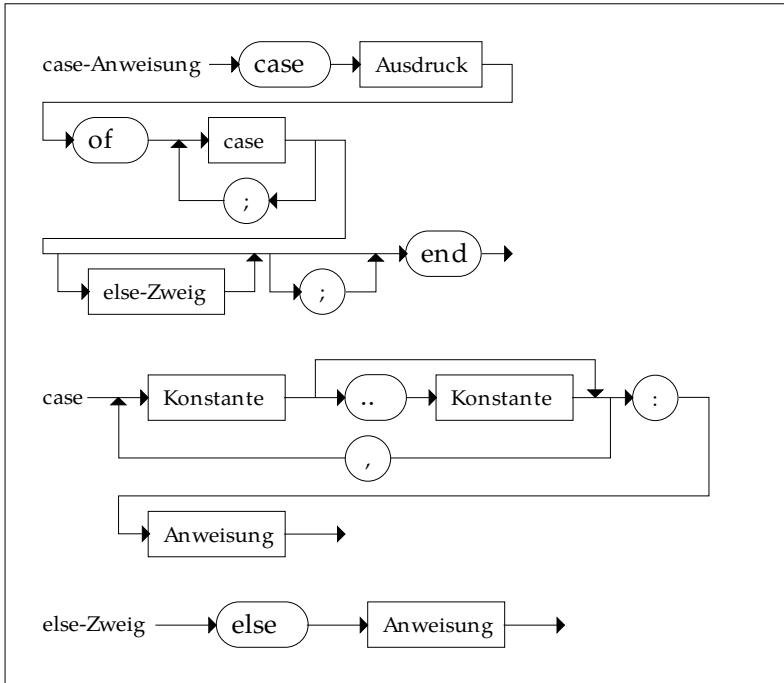


Abbildung 7.10 Syntax der case-Anweisung

Einige Beispiele:

case operator of

plus: X := X+Y;

minus: X := X-Y;

mal: **begin** X := X*Y; Y := 0 **end**
end;

case I of

0,2,4,6,8 : Writeln('Gerade Zahl');

1,3,5,7,9 : Writeln('Ungerade Zahl');

10..100 : Writeln('Zwischen 10 und 100');

else

Writeln('Der Wert ist entweder negativ');

Writeln(' oder größer als 100.');

end;

Schleifen

Schleifen legen die wiederholte Ausführung bestimmter Programmteile fest.

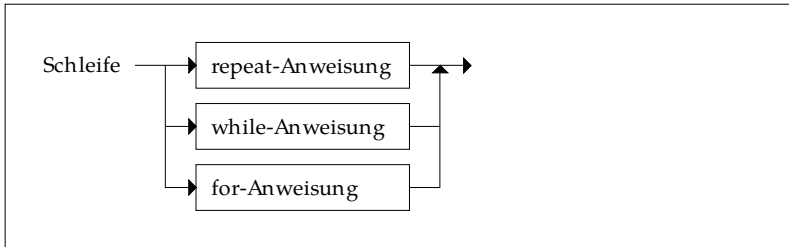


Abbildung 7.11 Syntax einer Schleifenanweisung (Wiederholungsanweisung)

Wenn die Anzahl der Wiederholungen von vornherein bekannt ist, sollte eine Konstruktion mit **for** verwendet werden – **repeat** und **while** sind hauptsächlich für Fälle gedacht, in denen die Abbruchbedingung durch Operationen innerhalb des Schleifenrumpfes gesetzt wird.

Die repeat-Anweisung

enthält einen Ausdruck, dessen Wahrheitswert festlegt, ob der durch **repeat..until** eingeschlossene Block wiederholt werden soll.

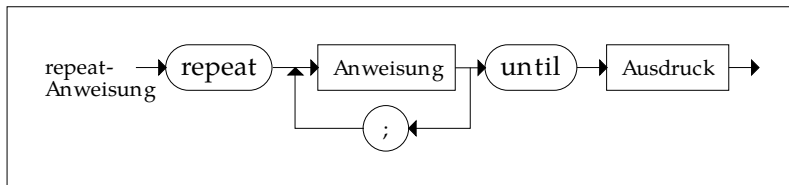


Abbildung 7.12 Syntax der repeat-Anweisung

Das Ergebnis des Ausdrucks muß vom Typ *Boolean* sein. Die Anweisungen zwischen **repeat** und **until** werden solange nacheinander ausgeführt, bis der Ausdruck hinter **until** den Wert *True* annimmt. Die Anweisungsfolge wird mindestens einmal ausgeführt, weil der Ausdruck erst nach Erreichen von **until** bewertet wird. Zwei Beispiele zu **repeat**:

repeat

```
K := I mod J;  
I := J;  
J := K;
```

```
until J = 0;
```

repeat

```
write('Geben Sie einen Wert von 0 bis 9 ein: ');  
Readln(I);
```

```
until (I >= 0) and (I <= 9);
```

Die while-Anweisung

enthält einen Ausdruck, dessen Wahrheitswert bestimmt, ob die auf **while** folgende einfache bzw. Verbundanweisung wiederholt ausgeführt wird oder nicht.

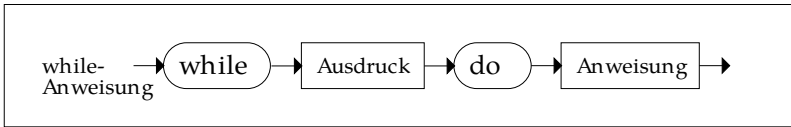


Abbildung 7.13 Syntax der while-Anweisung

Wie bei **repeat** muß der Ausdruck den Typ *Boolean* haben. Er wird jeweils *vor* der Ausführung des zu wiederholenden Programnteils bewertet – wenn sich bereits vor der ersten Ausführung der Wert *False* ergibt, wird der auf **while** folgende Block überhaupt nicht ausgeführt. Ansonsten wird er solange wiederholt, bis der Ausdruck den Wert *False* annimmt.

Der wesentliche Unterschied zwischen **while** und **repeat** liegt also darin, daß **while** eine Schleife wiederholt, *solange* der Ausdruck *True* ergibt; **repeat** wiederholt die Schleife, *bis* der Ausdruck den Wert *True* annimmt.

Einige Beispiele zu **while**:

```
while Daten[I] <> x do I := I + 1;
```

```
while I > 0 do
```

```
begin
```

```
  if Odd(I) then Z := Z * X;
```

```
  I := I div 2;
```

```
  X := Sqr(X);
```

```
end;
```

```
while not Eof(Eingabedatei) do
```

```
begin
```

```
  Readln(Eingabedatei, Zeile);
```

```
  Bearbeiten(Zeile);
```

```
end;
```

Die for-Anweisung

führt eine einfache oder eine Verbundanweisung in einer Schleife aus, wobei einer Variablen (der Laufvariablen) fortlaufende Werte zugewiesen werden.

Die Laufvariable muß der Bezeichner einer Variablen ordinalen Typs sein, die entweder innerhalb desselben Blocks wie die **for**-Anweisung lokal oder global zum gesamten Programm deklariert ist. Die Definition einer Schleife mit **for** schließt die Festlegung eines Start- und Endwertes mit ein. Beide Werte müssen einen Ordinal-Typ haben, der zu dem der Laufvariablen zuweisungskompatibel ist.

Beim Start der Schleife wird die Laufvariable auf den Startwert gesetzt und für jeden Schleifendurchlauf um eins erhöht bzw. erniedrigt – solange, bis der Endwert erreicht ist. Bei jedem Durchlauf wird die im Rumpf der Schleife

enthaltene Anweisung bzw. Verbundanweisung einmal ausgeführt. Wenn die Endbedingung der Schleife bereits vor dem ersten Durchlauf gegeben ist (d.h. Endwert < Startwert bzw. Endwert > Startwert bei Verwendung von **downto**), dann werden Schleife und dazugehöriger Rumpf komplett übersprungen.

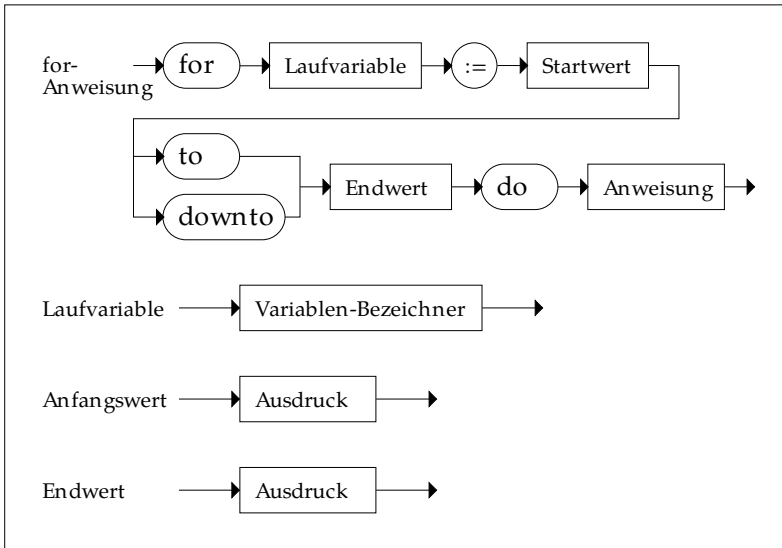


Abbildung 7.14 Syntax der for-Anweisung

Zuweisungen an die Laufvariable innerhalb einer **for**-Schleife sind unzulässig. Der Wert der Laufvariablen nach Ende der Schleife ist undefiniert – es sei denn, die Ausführung der Schleife wurde durch ein **goto** innerhalb des Rumpfes abgebrochen. Unter Beachtung dieser Einschränkungen führt die Anweisung

```
for V := Expr1 to Expr2 do
  Rumpf;
```

zum selben Ergebnis wie das Konstrukt:

```
begin
  Temp1 := Expr1;
  Temp2 := Expr2;
  if Temp1 <= Temp2 then
    begin
      V := Temp1;
      Rumpf;
      while V <> Temp2 do
        begin
          V := Succ(V);
          Rumpf;
        end;
    end;
```

```
end;
end;
```

Analoges gilt für **for**-Anweisungen mit **downto** – auch hier sind die beiden folgenden Konstrukte praktisch identisch:

```
for V := Expr1 downto Expr2 do
  Rumpf;
```

```
begin
  Temp1 := Expr1;
  Temp2 := Expr2;
  if Temp1 >= Temp2 then
    begin
      V := Temp1;
      Rumpf;
      while V <> Temp2 do
        begin
          V := Pred(V);
          Rumpf;
        end;
      end;
    end;
  end;
```

Temp1 und *Temp2* sind hier Hilfsvariablen des Grundtyps der Variablen *V* und erscheinen an keiner anderen Stelle im Programm. Einige weitere Beispiele für **for**-Anweisungen:

```
for I := 2 to 63 do
  if Daten[I] > Maximum then Maximum := Daten[I];

for I := 1 to 10 do
  for J := 1 to 10 do
    begin
      X := 0;
      for K := 1 to 10 do
        X := X + Mat1[I,K] * Mat2[K,J];
      Mat[I,J] := X;
    end;

for C := rot to blau do Farbwahl(C);
```

Die with-Anweisung

verkürzt den Bezug auf Felder von Records sowie auf Komponenten von Objekten (Felder, Methoden, Konstruktoren und Destruktoren). Zur Vereinfachung ist im folgenden Text ausschließlich von Record-Feldern die Rede, obgleich sowohl Record-Felder als auch Objekt-Komponenten gemeint sind.

Innerhalb einer **with**-Anweisung kann auf die Felder einer oder mehrerer Record-Variablen Bezug genommen werden, wobei lediglich die Angabe der Feldbezeichner notwendig ist.

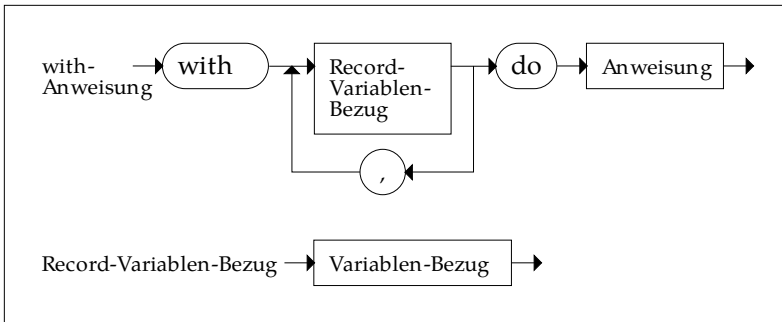


Abbildung 7.15 Syntax der with-Anweisung

Die beiden folgenden Beispiele sind also identisch:

```

with Datum do
  if Monat = 12 then
    begin
      Monat := 1;
      Jahr := Jahr + 1;
    end else
      Monat := Monat + 1;
  if Datum.Monat = 12 then
    begin
      Datum.Monat := 1;
      Datum.Jahr := Datum.Jahr + 1;
    end else
      Datum.Monat := Datum.Monat + 1;
  
```

Innerhalb des **with**-Blocks wird jeder Variablenbezug daraufhin geprüft, ob er als Feld eines Records behandelt werden kann. Fällt diese Prüfung positiv aus, wird die entsprechende Variable immer als Feld interpretiert – selbst dann, wenn auch auf eine andere Variable gleichen Namens zugegriffen werden könnte.

Das folgende Beispiel verdeutlicht diese Problematik. Nach einer Deklaration wie

```

type
  Point = record
    x,y: Integer;
  end;

var
  x: Point;
  y: Word;
  
```

können sich x und y – je nach Kontext – entweder auf Variablen oder Felder des Records beziehen, also:

```
with x do
  begin
    x := 10;    {Feld x des Records}
    y := 20;    {Feld y des Records}
  end;
y := 20;      {Word-Variable y}
```

with-Anweisungen können mehrfach verschachtelt werden. Die folgenden Zeilen sind identisch:

```
with V1, V2, ... Vn do <Anweisung>;

with V1 do
  with V2 do
    ...
    with Vn do
      <Anweisung>;
```

Wenn V_n ein Feld von V_1 und von V_2 ist, wird es in beiden Fällen als $V_2.V_n$ und nicht als $V_1.V_n$ interpretiert. Wenn die Adressierung einer Record-Variablen die Indizierung eines Arrays oder die Ermittlung einer Zeiger-Adresse erfordert, wird die entsprechende Aktion vor dem Beginn des Anweisungsteils ausgeführt.

8. Prozeduren und Funktionen

Prozeduren und Funktionen stellen formal gesehen zusätzliche Ebenen innerhalb des Hauptprogramm-Blocks dar, also eine Verschachtelung. Eine Prozedur wird durch einen Prozeduraufruf (d.h. die Angabe ihres Bezeichners) aktiviert und liefert keinen direkten Wert zurück. Eine Funktion wird bei der Berechnung eines Ausdrucks aktiviert, in der ihr Bezeichner erscheint, und hat normalerweise ein Ergebnis, das für diesen Aufruf mit dem Funktionsbezeichner gleichgesetzt werden kann.

Dieses Kapitel erläutert die verschiedenen Typen von Prozedur- und Funktionsdeklarationen sowie dazugehörige Parameter.

Prozedurdeklarationen

Eine Deklaration, die mit dem reservierten Wort **procedure** eingeleitet wird, verbindet einen Bezeichner und einen Block von Anweisungen zu einer Prozedur. Derartig deklarierte Prozeduren können durch Angabe ihres Bezeichners aktiviert (d.h. aufgerufen) werden.

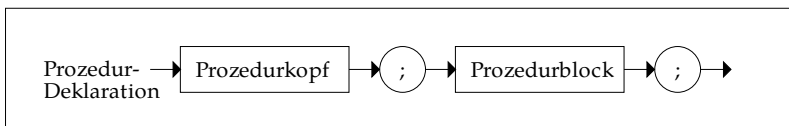


Abbildung 8.1 Syntax einer Prozedurdeklaration

Der Prozedurkopf benennt die Prozedur (d.h. ordnet ihr einen Bezeichner zu) und deklariert formale Parameter (soweit vorhanden).

Die Syntax einer formalen Parameterliste wird im Abschnitt »Parameter« in diesem Kapitel besprochen. Durch die Angabe ihres Bezeichners sowie einer Liste aktueller Parameter wird eine Prozedur *aktiviert*: Die im Anweisungsteil der entsprechenden Prozedurdeklaration definierten Aktionen werden ausgeführt.

Eine Prozedur, die ihre eigene Aktivierung als Bestandteil ihres Anweisungsteils enthält, wird rekursiv ausgeführt, das heißt, sie ruft sich selber wiederholt auf.

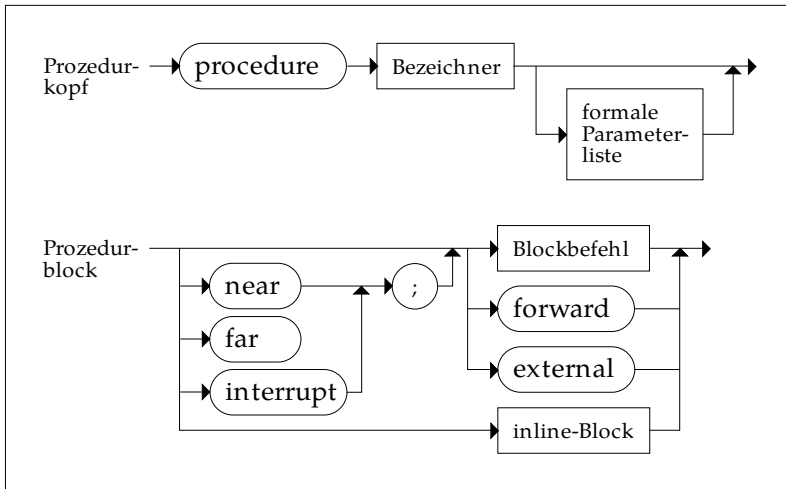


Abbildung 8.2 Syntax eines Prozedurkopfes

Ein Beispiel für eine Prozedurdeklaration:

```
procedure NumString(N:Integer; varS: string);
var
  v: Integer;   {lokale Variable}
begin
  V := Abs(N);
  S := '';
  repeat
    S := Chr(N mod 10 + Ord('0')) + S;
    N := N div 10;
  until N = 0;
  if N < 0 then S := '-' + S;
end;
```

Near- und far-Deklarationen

Turbo Pascal unterstützt zwei Modelle von Prozedur-Aufrufen: **near** und **far**. Bezüglich Codegröße und Ausführungsgeschwindigkeit ist das **near**-Modell dem **far**-Modell überlegen, aber es besitzt die Einschränkung, daß die Prozeduren nur von dem Modul aus aufgerufen werden können, in dem sie deklariert wurden. **far**-Prozeduren hingegen können von überall angesprochen werden, aber der Code für **far**-Aufrufe ist etwas langsamer und größer als der von **near**-Aufrufen.

Aufgrund der Prozedur-Deklaration wählt Turbo Pascal automatisch das richtige Aufruf-Modell aus: Prozeduren, die im **interface**-Teil einer Unit dekla-

riert wurden, benutzen das **far**-Modell. Prozeduren, die nur im **implementa-tion**-Teil einer Unit auftauchen, erhalten das **near**-Modell. Sie können ohnehin nur innerhalb der Unit bzw. des Programmes aufgerufen werden.

In einigen speziellen Fällen werden Prozeduren benötigt, die das **far**-Modell benutzen: In einer Applikation, die Overlays enthält, müssen alle Prozeduren und Funktionen das **far**-Modell gebrauchen. Außerdem können nur **far**-Prozeduren einer Prozedur-Variablen zugewiesen werden. Die Compiler-Direktive `$F` kann eingesetzt werden, um die automatische Modell-Auswahl des Compilers zu überschreiben.

Prozeduren und Funktionen, die im `{ $F$ +}`-Modus übersetzt wurden, benutzen immer das **far**-Modell. Im voreingestellten `{ $F$ -}`-Modus wählt der Compiler automatisch das richtige Aufruf-Modell.

Um für eine Prozedur oder Funktion ein ganz bestimmtes Aufruf-Modell zu setzen, kann vor dem Prozedur-Block eine **far**- bzw. **near**-Direktive geschrieben werden. Eine solche Direktive überschreibt sowohl die `$F`-Direktive als auch die automatische Wahl des Aufruf-Modells. Ein Beispiel:

```
procedure ExitApplication; far;
begin
  Application^.Done;
end;

begin
  ExitProc:= @ExitApplication;
end.
```

Deklarationen mit interrupt

In Turbo Pascal kann eine Prozedurdeklaration mit dem ebenfalls optionalen Zusatz **interrupt** versehen werden, der direkt auf den Prozedurkopf folgt. Eine vollständige Besprechung derartiger Prozeduren finden Sie in Kapitel 14 »Interne Details« – wir wollen an dieser Stelle nur auf die formalen Voraussetzungen eingehen:

■ **interrupt**-Prozeduren können nicht wie andere Prozeduren über ihren Namen aufgerufen werden, sondern nur über Interrupt-Vektoren des Systems.

■ Der Kopf einer als **interrupt** deklarierten Prozedur hat ein fixes Format mit einer festgelegten Anzahl und Reihenfolge von Parametern, nämlich:

```
procedure <Name> (Flags, CS, IP, AX, BX, CX, DX, SI, DI, DS, ES, BP: Word);
interrupt;
```

Auf den Kopf einer »normalen« Prozedur kann anstelle eines Blocks auch eine der Direktiven **forward**, **external** oder **inline** folgen. Zweck und Wirkung dieser Deklarationen werden in den folgenden Abschnitten besprochen.

Deklarationen mit forward

Eine Prozedurdeklaration, auf die anstelle eines Vereinbarungs- und Anweisungsteils das reservierte Wort **forward** folgt, wird als **forward**-Deklaration bezeichnet. Der eigentliche Prozedurblock muß zu einem späteren Zeitpunkt (d.h. weiter »unten« im Programmtext) definiert werden. Diese Definition beginnt wiederum mit **procedure**, gefolgt vom Bezeichner der als **forward** deklarierten Prozedur, enthält jedoch keine formale Parameterliste.

Zwischen der **forward**-Deklaration und der Definition der Prozedur ist die Deklaration weiterer Prozeduren und Funktionen möglich, die ihrerseits die als **forward** deklarierte Prozedur aufrufen können. Auf diese Weise ist der sich überkreuzende Bezug zwischen zwei oder mehreren Prozeduren herstellbar (d.h. die Definition von Programmteilen, die sich gegenseitig aufrufen).

Hinweis: Die **forward**-Deklaration und die Definition der Prozedur müssen in ein und demselben Deklarationsteil des Programms stehen. In einem Konstrukt wie

```

procedure FF; forward;   { forward-Deklaration }
....
type                    { eingeschobener Typen-Definitionsteil }
....

procedure FF;           { Definition der Prozedur }
begin
...

```

wird der durchgehende Deklarationsteil des Programms von einem Typen-Definitionsteil unterbrochen – folglich meldet der Compiler einen Fehler. Analoges gilt für »Unterbrechungen« durch die Deklaration globaler Variablen und/oder Konstanten.

Die **forward**-Deklaration bildet zusammen mit dem Prozedurblock eine vollständige Prozedur. Die Prozedur selber gilt mit **forward** als deklariert. Ein Beispiel:

```

procedure Walter(m,n: Integer); forward;

procedure Klara(x,y: Real);
begin
....
  Walter(4,5);
....
end;

procedure Walter;          {Definition der forward-Prozedur}
begin
....
  Klara(8.3,2.4);
....
end;

```

Die Definition einer als **forward** deklarierten Prozedur kann als **external** oder als **assembler** geschehen – **far**, **near**, **inline**, **interrupt** oder die Definition über ein weiteres **forward** sind dagegen nicht zulässig.

Im Interface-Teil einer Unit sind **forward**-Deklarationen weder notwendig noch erlaubt.

Deklarationen mit external

ermöglichen die Einbindung (das »Linken«) getrennt kompilierter Prozeduren und Funktionen, die in Maschinensprache geschrieben worden sind. Der externe Code wird über den Compiler-Befehl `{L Dateiname}` mit einem Pascal-Programm oder einer Unit verbunden.

Einige Beispiele dazu:

```
procedure MoveWord(var Qu, Zi; Zahl: LongInt); external;
procedure MoveLong(var Qu, Zi; Zahl: LongInt); external;
procedure FillW(var Ziel; Data: Integer; Z: LongInt); external;
procedure FillL(var Ziel; Data: LongInt; Z: LongInt); external;
```

`{L Blkstuff.obj}` {<- eine selbstgeschriebene Datei}

Weitere Details zu **external**-Prozeduren finden sich in Kapitel 15 »Der integrierte Assembler und Einbinden von Assembler-Code«.

Deklarationen mit assembler

Mit Hilfe einer **assembler**-Deklaration können ganze Prozeduren und Funktionen mit dem integrierten Assembler geschrieben werden.

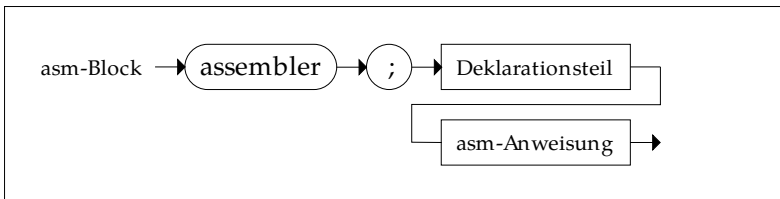


Abbildung 8.3 Syntax einer assembler-Deklaration

In Kapitel 15 »Der integrierte Assembler und Einbinden von Assembler-Code« finden Sie weitere Informationen zu **assembler**-Deklarationen

Deklarationen mit inline

machen es möglich, anstelle eines Pascal-Anweisungsblocks Maschinencode zu verwenden: Nach dem reservierten Wort **inline** erwartet der Compiler eine Liste von Konstanten (normalerweise in hexadezimaler Notation), die direkt in das Programm aufgenommen werden.

Im Normalfall generiert der Compiler bei einem Prozeduraufruf Code, der die Argumente der Prozedur auf den Stack legt, danach erzeugt er einen CALL-Befehl zum Aufruf der Prozedur. Wenn eine **inline**-Prozedur »aufgerufen« wird, werden die Konstanten, die nach **inline** definiert sind, vom Compiler ohne einen CALL direkt in den laufenden Code eingesetzt. Jede Konstante bildet genau ein Wort im generierten Code, der Compiler bildet bei der Codegenerierung die Reihenfolge der Konstanten ab.

inline sollte anstelle von **external** zur Implementation kleinerer Routinen (mit maximal 10 oder 20 Prozessorbefehlen) verwendet werden. Zwei Beispiele für **inline**-Prozeduren:

```
procedure Interrupts_An; inline ($FA); { CLI }  
procedure Interrupts_Aus; inline ($FB); { STI }
```

Eine vollständige Besprechung von **inline**-Prozeduren finden Sie in Kapitel 15 »Der integrierte Assembler und Einbinden von Assembler-Code«.

Funktionsdeklarationen

Eine Funktionsdeklaration definiert einen Programmteil, der einen Wert berechnet und als Funktionsergebnis zurückliefert.

Der Funktionskopf legt den Bezeichner, die formalen Parameter (soweit vorhanden) und den Ergebnistyp der Funktion fest (siehe Abb. 8.4).

Funktionen werden bei der Berechnung von Ausdrücken aktiviert, in denen ihr Bezeichner erscheint, und stehen dort stellvertretend für den von ihnen zurückgelieferten Wert. Ein Funktionsbezeichner kann überall anstelle eines Operanden in einem Ausdruck eingesetzt werden, solange der Typ des Funktionsergebnisses mit dem des ersetzten Operanden kompatibel ist. Zuweisungen an einen Funktionsbezeichner sind nur innerhalb der Funktion selber erlaubt.

Der Aufruf einer Funktion geschieht über die Angabe ihres Bezeichners, gefolgt von einer Liste aktueller Parameter, die in Typ und Reihenfolge den formalen Parametern der Funktionsdeklaration entsprechen müssen.

Der Anweisungsblock einer Funktion legt alle Operationen fest, die nach der Aktivierung der Funktion ausgeführt werden sollen. Dieser Block sollte zumindest eine Zuweisung enthalten, die dem Funktionsbezeichner einen Wert gibt. Das Ergebnis der Funktion ist der jeweils letzte zugewiesene Wert vor dem Ende des Anweisungsteils der Funktion. Existiert eine solche Zuweisung nicht, oder


```

function Potenz(x: Extended; y: Integer): Extended;
var
  z: Extended;
  i: Integer;
begin
  z := 1.0; i:= y;
  while i > 0 do begin
    If Odd(i) then z := z * x;
    i := i div 2;
    x := Sqr(x);
  end;
  Potenz := z;
end;

```

Funktionen können in derselben Weise wie Prozeduren als **forward**, **external**, **near**, **far** oder **inline** deklariert werden – der Zusatz **interrupt** ist dagegen nicht erlaubt.

Deklarationen von Methoden

Die Deklaration einer Methode innerhalb eines Objekttyps entspricht einer **forward**-Deklaration dieser Anweisung. Deshalb muß die Methode nach der Deklaration des Objekttyps im selben Gültigkeitsbereich mit Hilfe einer Definition *implementiert* werden.

Werden Prozeduren und Funktionen als Methoden definiert, so hat diese Definition die Form einer normalen Prozedur- oder Funktionsdeklaration, allerdings mit der Ausnahme, daß in diesem Fall als Prozedur- oder Funktionsbezeichner ein vollständiger Methodenbezeichner verwendet werden muß.

Die Definition von Konstruktoren und Destruktoren hat dieselbe Form wie die Definition einer Methode, die als Prozedur deklariert wird, außer, daß das Schlüsselwort **procedure** durch die reservierten Wörter **constructor** und **destructor** ersetzt wird.

Die Definition einer Methode kann optional die formale Parameterliste des Deklarationskopfs der Methode im Objekttyp wiederholen. Die Kopfzeile der Definition muß in diesem Fall in Bezug auf Reihenfolge, Typ und Bezeichner sowie gegebenenfalls dem Funktionsresultat exakt übereinstimmen.

In der Definition einer Methode ist immer automatisch die Bezeichnung *Self* deklariert, die formal einem Variablenparameter vom Typ des jeweiligen Objekts entspricht. Innerhalb des Anweisungsblocks stellt *Self* die Instanz dar, über deren Methode die Methode aktiviert wurde. Deshalb wirken sich alle Veränderungen der Felder von *Self* auch in der Instanz selbst aus.

Der Gültigkeitsbereich einer Komponentenbezeichnung in einem Objekttyp reicht über alle Anweisungsblöcke von Prozeduren, Funktionen, Konstruktoren und Destruktoren, in denen eine Methode des Objekttyps implementiert ist.

Das bedeutet dasselbe, als stünde der gesamte Methodenblock in einer **with**-Anweisung der Form

```
with Self begin ... end
```

Aus diesem Grund muß die Schreibweise von Komponentenbezeichnungen, formalen Parametern, *Self* und allen lokalen Variablen in einer Methode eindeutig sein. Nachfolgend sehen Sie nun einige Beispiele für die Implementierung von Methoden:

```
procedure Rect.Intersect(var R : Rect);
begin
  if A.X < R.A.X then A.X := R.A.X;
  if A.Y < R.A.Y then A.Y := R.A.Y;
  if B.X > R.B.X then B.X := R.B.X;
  if B.Y > R.B.Y then B.Y := R.B.Y;
  if (A.X >= B.X) or (A.Y >= B.Y) then Init(0, 0, 0, 0);
end;

procedure Field.Display;
begin
  GotoXY(X, Y);
  Write(Name^, ' ', GetStr);
end;

function NumField.PutStr(S : String) : Boolean;
var
  E: Integer;
begin
  Val(S, Wert, E);
  PutStr := (E = 0) and (Value >= Min) and (Value <= Max);
end;
```

Konstrukturen und Destrukturen

Konstrukturen und Destrukturen sind zwei spezielle Formen von Methoden. In Verbindung mit der erweiterten Syntax der Standardprozeduren *New* und *Dispose* kann man mit einem Konstruktor dynamische Objekte im Speicher anlegen und mit einem Destruktor den durch die Objekte belegten Speicherplatz wieder freigeben. Zusätzlich bieten Konstrukturen die Möglichkeit, die erforderliche Initialisierung der Objekte, in denen virtuelle Methoden enthalten sind, vorzunehmen. Wie andere Methoden können Konstrukturen und Destrukturen vererbt werden. Ein Objekt kann beliebig viele Konstrukturen und Destrukturen enthalten.

Konstrukturen werden zur Initialisierung von Instanzen eines Objekts verwendet. Normalerweise funktioniert die Initialisierung so, daß die Initialisierungswerte als Parameter an den Konstruktor übergeben werden. Konstrukturen dürfen nicht virtuell sein, weil der Mechanismus zum Ermitteln der Adresse

einer virtuellen Methode in der VMT von einem Konstruktor abhängt, mit dem das Objekt vorher bereits initialisiert worden sein muß.

Hier sehen Sie nun einige Beispiele für Konstruktoren:

```
constructor Field.Copy(var F :Field);
begin
    Self := F;
end;

constructor Field.Init(FX, FY, FLen : Integer; FName : String);
begin
    X := FX;
    Y := FY;
    Len := FLen;
    GetMem(Name, Length(FName) + 1);
    Name^ := FName;
end;

constructor StrField.Init(FX, FY, FLen : Integer; FName : String);
begin
    Field.Init(FX, FY, FLen, FName);
    GetMem(Value, FLen);
    Value^ := '';
end;
```

Die erste Aktion des Konstruktors eines Nachkommens, wie zum Beispiel des oben dargestellten *StrField.Init*, ist fast immer der Aufruf des entsprechenden Konstruktors des direkten Vorfahren, um die übernommenen Felder des Objekts zu initialisieren. Nachdem das geschehen ist, initialisiert der Konstruktor die Felder des Objekts, die vom Nachkommen eingeführt wurden.

Destruktoren sind die Gegenstücke zu den Konstruktoren und werden zum Aufräumen der Objekte nach erfolgter Anwendung verwendet. Normalerweise besteht das Aufräumen in der Freigabe aller Zeigerfelder in einem Objekt.

Im Gegensatz zu Konstruktoren können Destruktoren virtuell sein. Sie besitzen üblicherweise keine formalen Parameter.

Hier sehen Sie einige Beispiele für Destruktoren:

```
destructor Field.Done;
begin
    FreeMem(Name, Length(Name^) + 1);
end;

destructor StrField.Done;
begin
    FreeMem(Value, Len);
    Field.Done;
end;
```

Ein Destruktor eines Nachkommens, wie zum Beispiel des oben aufgeführten *StrField.Done*, gibt normalerweise zuerst die vom Nachkommen eingeführten Zeigerfelder frei und ruft dann als letzte Aktion den entsprechenden Destruktor

Wert- parameter

Ein formaler Wertparameter verhält sich wie eine lokale Variable der entsprechenden Prozedur bzw. Funktion – der einzige Unterschied besteht darin, daß er beim Aufruf den Wert eines aktuellen Parameters erhält. Veränderungen von Wertparametern beeinflussen den Wert des aktuellen Parameters nicht.

Der korrespondierende aktuelle Parameter eines formalen Parameters kann – im Gegensatz zu Variablenparametern – ein Ausdruck sein. Der Wert dieses Ausdrucks darf weder einen Dateityp noch einen strukturierten Typ haben, der einen Dateityp als Komponente enthält.

Der aktuelle Parameter muß zu seinem formalen Parameter zuweisungskompatibel sein. Wenn der Parameter den Typ **string** hat, wird seine Maximallänge automatisch auf 255 Zeichen festgelegt.

Variablen- parameter

Ein Variablenparameter wird benutzt, wenn ein Wert einer Prozedur oder Funktion durch Bezugnahme übergeben werden soll. Der korrespondierende aktuelle Parameter innerhalb eines Prozedur- oder Funktionsaufrufs muß ein Variablenbezug sein – die Übergabe von Ausdrücken als Variablenparameter ist also nicht zulässig. Der formale Variablenparameter repräsentiert während der Aktivierung der Prozedur oder Funktion den aktuellen Parameter. Jede Veränderung des formalen Variablenparameters beeinflußt auch den Wert des aktuellen Parameters (d.h. der übergebenen Variablen).

Innerhalb der Prozedur oder Funktion wird mit jedem Bezug auf den formalen Parameter auf den Wert des aktuellen Parameters zugegriffen. Beide Parameter müssen denselben Typ haben – eine Einschränkung, die durch die Verwendung untypisierter Parameter umgangen werden kann. Hat der formale Parameter den Typ **string**, erhält er das Längenattribut 255 – der aktuelle Variablenparameter muß in diesem Fall dasselbe Längenattribut besitzen.

Dateitypen können ausschließlich als Variablenparameter übergeben werden. Operationen wie die Indizierung eines Arrays oder die Berechnung einer Zeigeradresse für den Bezug auf aktuelle Variablenparameter werden vor der Übergabe des Parameters ausgeführt.

Für Variablenparameter mit Objekttypen gelten die Regeln für die Kompatibilität von Zuweisungen zwischen Objekten (vgl. Kapitel 7 »Anweisungen«). Für einen formalen Parameter des Typs *T1* kann der aktuelle Parameter den Typ *T2* haben, wenn *T2* im Bereich von *T1* liegt. Zum Beispiel kann der Methode *Field.Copy* eine Instanz von *Field*, *StrField*, *NumField*, *ZipField* oder eine andere Instanz eines Nachkommens von *Field* übergeben werden.

Untypisierte Variablenparameter

Wenn der Typ eines formalen Parameters nicht definiert ist, d.h. dem dazugehörigen Bezeichner im Prozedur- bzw. Funktionskopf kein Typ-Bezeichner folgt, dann wird dieser Parameter als untypisiert bezeichnet. In diesem Fall kann der entsprechende aktuelle Parameter jeden beliebigen Typ haben.

Untypisierte Variablen sind inkompatibel zu allen anderen Variablen - schließlich haben sie keinen Typ. Deshalb muß ihnen vor ihrer Verwendung über eine Typumwandlung ein spezifischer Typ zugewiesen werden. Ein Beispiel für eine Funktion, die mit zwei untypisierten Parametern arbeitet:

```
function Gleich(var Quelle, Ziel; Zahl : Integer): Boolean
type
  Bytes = array[0..MaxInt] of Byte;
var
  N: Integer;
begin
  N := 0;
  while (N < Zahl) and (Bytes(Quelle)[N] <> Bytes(Ziel)[N]) do
    Inc(N);
  Gleich := N = Zahl;
end;
```

Mit dieser Funktion können zwei beliebige Variablen jeder Größe miteinander verglichen werden. Gegeben seien die folgenden Deklarationen und Funktionsaufrufe:

```
type
  Vector = array[1..10] of Integer;
  Point = record
    x,y: Integer;
  end;

var
  Vec1, Vec2: Vector;
  N: Integer;
  P: Point;
  B: Boolean;

begin
  B := Gleich(Vec1,Vec2,SizeOf(Vector));
  WriteLn(B);
  B := Gleich(Vec1,Vec2,Sizeof(Integer)*N);
  WriteLn(B);
  B := Gleich(Vec[1],Vec1[6],SizeOf(Integer)*5);
  WriteLn(B);
  B := Gleich(Vec1[1],P,4); WriteLn(B);
end.
```

Der erste Aufruf vergleicht sämtliche Elemente von *Vec1* mit denen von *Vec2*; der zweite vergleicht nur die ersten N Komponenten beider Arrays. Der dritte Aufruf vergleicht die ersten fünf Komponenten von *Vec1* mit den Komponenten 6..10 von *Vec2*, der vierte vergleicht schließlich *Vec1*[1] mit dem Feld *P.x* und *Vec1*[2] mit dem Feld *P.y*.

Prozedurtypen

Als Erweiterung des Sprachstandards ermöglicht Turbo Pascal die Behandlung von Prozeduren und Funktionen als *Objekte* des Programms (und nicht als unveränderlicher Teil von ihm). Eine neue Klasse von Variablen – die *Prozedur-Variablen* – speichert nicht nur (wie Zeiger) die Adresse einer Routine, sondern hat für den Compiler auch die restlichen Charakteristika einer Prozedur bzw. Funktion. Eine detaillierte Erklärung von Prozedurtypen finden Sie in Kapitel 3 »Typen«.

Prozedur-Variablen

Nach der Vereinbarung eines Prozedurtyps lassen sich entsprechende Variablen deklarieren – wenn wir von den Definitionen des letzten Abschnitts ausgehen, beispielsweise:

```
var
  SwapProcPtr: SwapProc;
  F: MathFunc;
  X, Y: Integer; { zwei "normale" Variablen }
```

Technisch gesehen, unterscheiden sich derartige Variablentypen in keinem Punkt von normalen Zeigern: sie enthalten eine Adresse und belegen vier Bytes im Speicher.

Die Fortsetzung unseres Beispiels:

```
procedure Swap(var A, B: Integer); far;
var
  Temp: Integer;
begin
  Temp := A; A := B; B:= Temp;
end;

function Tan(Winkel: Real): Real; far;
begin
  Tan:= Sin(Winkel) / Cos(Winkel);
end;
```

begin

```

SwapProcPtr := Swap; { Zuweisung an SwapProcPtr }
SwapProcPtr(X,Y);    { Aufruf der Prozedur Swap }
Swap(X,Y);           { dito }
F := Tan;             { Zuweisung an Tan }
WriteLn(F(X));       { Aufruf der Funktion Tan }

```

end.

Abgesehen davon, daß der Aufruf über *SwapProcPtr* mit einer normalen Pascal-Anweisung erfolgt, erschöpfen sich die Unterschiede zu Zeigern des Typs *Pointer* in zwei Punkten:

- in der Art der Zuweisung, die *direkt* (also ohne den Adreß-Operator @) erfolgt;
- und in der Typisierung: eine Prozedur-Variable enthält für den Compiler nicht »irgendeine« Adresse, sondern dieselben Charakteristika wie die Deklaration einer Prozedur bzw. Funktion.

Aufgrund der zweiten Eigenschaft lassen sich hier dieselben Prüfmechanismen wie bei »normalen« Aufrufen anwenden – das Ergebnis einer Anweisung wie

```
SwapProcPtr(X);
```

ist die Meldung des Compilers, daß hier ein Parameter zuwenig angegeben wurde.

Dieselbe Prüfung wird bei Zuweisungen an Prozedur-Variablen vorgenommen: »Zuweisungskompatibilität« bedeutet hier, daß Routine und Variable dieselbe Anzahl von Parametern mit demselben Typ und in derselben Reihenfolge sowie (bei Funktionen) denselben Ergebnistyp deklarieren müssen.

Technische Voraussetzungen

Damit sich eine Routine einer Prozedur-Variablen zuweisen läßt, *muß* sie als **far** deklariert sein (was sich auch über den Compiler-Befehl `{SF+}` im Quelltext erreichen läßt). Die folgenden Arten von Routinen lassen sich *nicht* an Prozedur-Variablen zuweisen bzw. über Prozedur-Variablen aufrufen:

- eingeschachtelte Prozeduren und Funktionen (weil sie nicht global verfügbar sind);
- **inline**-Routinen (weil sie vom Compiler als »Makro« behandelt werden und deshalb keine Adresse haben);
- **interrupt**-Routinen (weil sie mit einem speziellen RET-Befehl enden, für den ein weiterer Prozedurtyp notwendig wäre);
- die Prozeduren und Funktionen der Unit *System*, d.h. Standardfunktionen wie *WriteLn*, *Read* (aufgrund ihrer variablen Parameterzahl), *Ord*, *Succ* usw. (weil sie vom Compiler direkt berechnet bzw. als direkte Befehle eingefügt werden).

Wenn Sie Standardfunktionen dennoch über Prozedur-Variablen aufrufen wollen, müssen Sie eine eigene Routine schreiben, die den Systemaufruf enthält und diese Routine einer Prozedur-Variablen zuordnen:

```

type
  IntProc = procedure(X: Integer);

var
  WriteProc: IntProc;

procedure WriteInt(Number: Integer); far;
begin
  Write(Number);
end;

begin
  WriteProc := WriteInt; { Zuordnung }
  WriteProc(10); { Aufruf von WriteInt über die Prozedur-Variable }
end.

```

Hinweis: Lokal definierte Routinen sind grundsätzlich als **near** codiert und lassen sich deshalb nicht über Prozedur-Variablen aufrufen. Der Versuch, nach einer Deklaration wie

```

procedure Aussen;

  procedure Innen;
  begin
    WriteLn("Innen" ist in "Außen" eingeschachtelt!");
  end; { Innen }

begin { Aussen }
  Innen;
end;

```

die Prozedur *Innen* einer Prozedur-Variablen zuzuordnen, scheitert am Widerstand des Compilers (der damit den Eigenheiten der Prozessorfamilie 80x86 bei *near*-Adressierungen Rechnung trägt).

Prozedur-Variablen in Strukturen

Da eine Prozedur-Variable (abgesehen von ihrer »Charakteristik« für den Compiler) letztendlich ein ganz normales Datum darstellt, läßt sie sich wie jede andere Variable auch als Element eines Arrays oder als Feld eines Records verwenden:

```

type
  GotoProc = procedure(X, Y: Integer);
  ProcList = array[1..10] of GotoProc;

  WindowPtr = ^Window;
  Window = record
    Next: WindowPtr;
    Title: string[20];
    Top, Left, Bottom, Right: Integer;
    SetCursor: GotoProc;
  end

```

var

```
P : ProcList;
W : WindowPtr;
```

Nach diesen Deklarationen sind die folgenden Prozeduraufrufe möglich:

```
P[3](1,1);           { ruft die Prozedur des Typs GotoProc auf, die
                      als drittes Element im Array P gespeichert
                      ist, und übergibt die Werte (1,1) }
W^.SetCursor(10,10); { ruft die Prozedur des Typs GotoProc auf,
                      die als Feld SetCursor eines Records des
                      Typs Window gespeichert ist, auf den W
                      zeigt, und übergibt die Werte (10,10) }
```

Prozedur-Variablen als Parameter

Die Behauptung, daß Prozedur-Variablen eigentlich völlig normale Daten darstellen, gilt nicht nur für die Aufnahme in Strukturen. Das folgende Programm demonstriert, wie sich einer Prozedur drei verschiedene Funktionen als *Parameter* übergeben lassen:

```
program Tables;

type
  Func = function(X,Y: Integer): Integer;

function Add(X,Y: Integer): Integer; far;
begin
  Add := X + Y;
end;

function Mult(A,B: Integer): Integer; far;
begin
  Mult := A * B;
end;

function Funny(X,Y: Integer): Integer; far;
begin
  Funny := (X+Y) * (X-Y);
end;
```

Diese drei Funktionen sollen über Prozedur-Variablen aufgerufen werden und müssen deshalb als **far** codiert sein; der Rest des Programms verwendet dagegen **near**-Aufrufe.

```
procedure PrintTable(W, H: Integer; Operation: Func);
var x,y: Integer;

begin
  for y:= 1 to H do
    begin
      for x := 1 to W do
        Write(Operation(X,Y):5);
      WriteLn;
```

```
end;
end;
```

PrintTable erwartet neben der Zeilen- und Spaltenzahl der Tabelle eine Prozedur-Variable des Typs *Func* als Parameter. Wie der Aufruf von *Write* zeigt, lassen sich Prozedur-Variablen auch bei Funktionen direkt anstelle eines Funktionsbezeichners verwenden: der Aufruf über *Operation* aktiviert die entsprechende Funktion und liefert deren Ergebnis zurück.

```
begin
  PrintTable(10,10,Add);
  PrintTable(10,10,Mult);
  PrintTable(10,10,Funny);
end.
```

Das Hauptprogramm ruft *PrintTable* dreimal auf, wobei jeweils eine andere Funktion als *Operation* übergeben wird. Der zweite Aufruf produziert beispielsweise das folgende Ergebnis:

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

Prozedur-Variablen als Parameter können Programme speziell dann wesentlich vereinfachen, wenn zwischen einer größeren Zahl ähnlicher Aktionen unterschieden werden muß – wie eben den Grundrechenarten oder der Bewegung des Cursors in einem Textverarbeitungsprogramm (was sonst über eine schier endlose Folge von **case**-Zweigen gelöst werden muß).

Als Parameter übergebene Prozedur-Variablen müssen den im Abschnitt »Technische Voraussetzungen« aufgelisteten Regeln genügen: Die übergebene Routine muß als **far** codiert und in ihrer Deklaration zuweisungskompatibel zum entsprechenden formalen Parameter sein; die Routinen von *System*, **inline**- und **interrupt**-Prozeduren sowie lokale Funktionen und Prozeduren lassen sich nicht als Parameter verwenden.

9. Programme und Units

Die Syntax eines Programms

Ein Turbo Pascal-Programm hat eine ähnliche Form wie eine Prozedurdeklaration. Die Unterschiede liegen lediglich im Programmkopf und der Möglichkeit einer zusätzlichen **uses**-Anweisung.

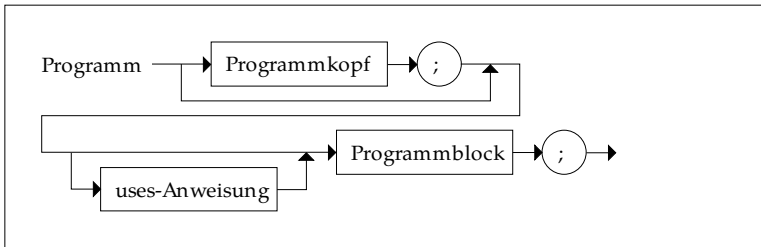


Abbildung 9.1 Syntax eines Turbo Pascal-Programms

Der Programmkopf

Der Programmkopf legt Namen und Parameter eines Programms fest.

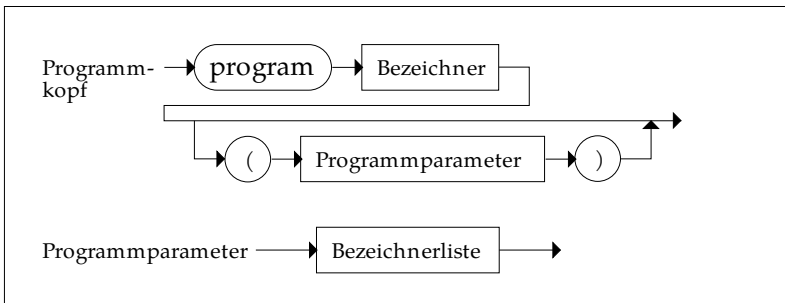


Abbildung 9.2 Syntax eines Programmkopfes

Programmparameter werden in Turbo Pascal vom Compiler ignoriert. Die folgenden Programmköpfe sind also identisch:

```
program Test(Input, Output);
program Test;
```

Die Anweisung *uses*

legt fest, welche Units ein Programm benutzt. Hier müssen nur direkt benutzte Units aufgeführt werden: Compiler und Linker prüfen bei der Aufnahme einer Unit automatisch, ob diese Unit ihrerseits andere Units voraussetzt, und binden sie gegebenenfalls in das Gesamtprogramm ein.

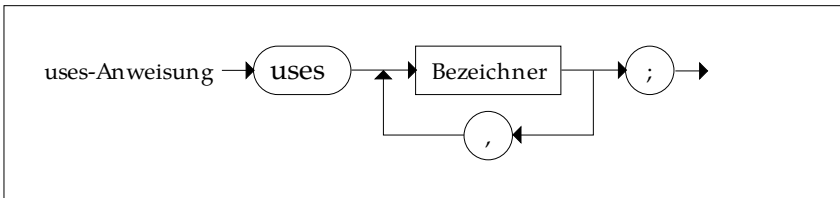


Abbildung 9.3 Syntax einer **uses**-Anweisung

Die Unit *System* wird automatisch vom Compiler in Programme eingefügt. Sie implementiert alle Laufzeitfunktionen des Systems auf niedriger Ebene, u.a. die Behandlung von Strings und Mengen, die dynamische Speicherbelegung und die Bereichsüberprüfung.

Darüber hinaus verfügt Turbo Pascal über eine Reihe von Standard-Units wie *Printer*, *Dos* und *Crt*. Ihre Aufnahme erfolgt nicht automatisch – sie müssen jeweils explizit über eine **uses**-Anweisung angegeben werden (vgl. Kapitel 12 »Standard-Units«):

```
uses Dos, Crt; { macht die Definitionen von DOS und CRT verfügbar }
```

Wenn eine Unit via **uses** angegeben wird, sucht der Compiler als erstes im Hauptspeicher, d.h. innerhalb der *residenten* Units, die beim Start von Turbo Pascal automatisch (aus der Datei TURBO.TPL) geladen wurden. Wenn sich die gesuchte Unit hier nicht findet, wird eine Datei mit dem Namen der Unit und dem Suffix .TPU auf der Diskette gesucht: zuerst innerhalb des momentan gesetzten Verzeichnisses, danach über die in *O/D/Unit Directories* bzw. den Kommandozeilen-Parameter */U* festgelegten Suchwege.

Wenn wir voraussetzen, daß die Unit *Memory* nicht in TURBO.TPL aufgenommen wurde, dann sucht der Compiler nach der folgenden Anweisung eine Datei namens MEMORY.TPU:

```
uses Memory;
```

Bei der Compilierung von (via **uses** angegebenen) Units über *Compile/Make* oder *Compile/Build* sucht Turbo Pascal die Quelltexte der entsprechenden Units

auf dieselbe Weise wie die .TPU-Dateien, nur daß hier das Suffix .PAS angenommen wird. Für die zuvor gezeigte **uses**-Anweisung wird also das Versionsalter der Dateien MEMORY.PAS und MEMORY.TPU verglichen bzw. die Datei MEMORY.PAS neu compiliert.

Die Reihenfolge der Unit-Namen in der **uses**-Anweisung bestimmt deren Initialisierungs-Reihenfolge (vgl. auch »Der Initialisierungs-Teil« in diesem Kapitel).

Die Syntax von Units

Units sind die Grundlage der modularen Programmierung in Turbo Pascal. Sie werden zur Erzeugung von Bibliotheken benutzt, die in verschiedene Programme eingebunden werden können, ohne den Quelltext selber zugänglich zu machen. Mit ihrer Hilfe läßt sich ein Programm in Module unterteilen, die nach der logischen Anordnung ihrer Bestandteile zusammengestellt werden.

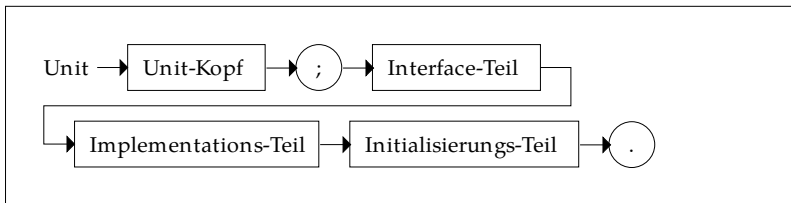


Abbildung 9.4 Syntax einer Unit

Der Kopf einer Unit

legt den Namen der Unit fest, der innerhalb des gesamten Programms »einzigartig« sein muß – mehrere Units mit demselben Namen sind innerhalb eines Programms nicht möglich.

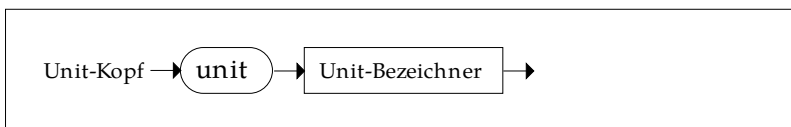


Abbildung 9.5 Syntax eines Unit-Kopfes

Der Interface-Teil

einer Unit deklariert alle »öffentlichen« Typen, Variablen, Prozeduren und

Funktionen. Die dazugehörigen Bezeichner sind für das Programm und jede Unit zugänglich, das diese Unit benutzt. Programme und Units können darauf in derselben Weise zugreifen, als wären sie in einem übergeordneten Block deklariert worden.

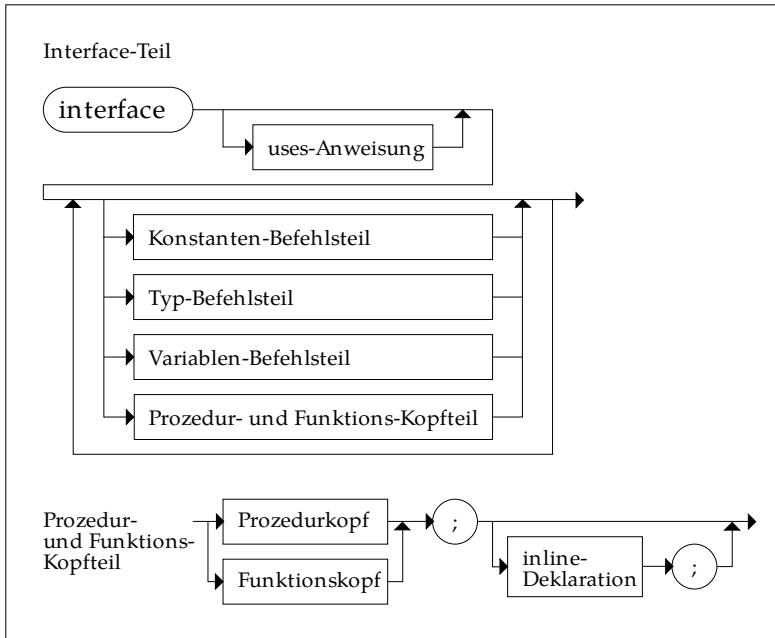


Abbildung 9.6 Syntax des Interface-Teils einer Unit

Solange eine Prozedur oder Funktion nicht als **inline** deklariert ist, enthält der Interface-Teil nur ihren Kopf. Der Block der Prozedur oder Funktion befindet sich im Implementations-Teil der Unit.

Der Implementations-Teil

definiert die Anweisungsblöcke (»Rümpfe«) aller »öffentlichen« Prozeduren und Funktionen, solange diese nicht als **inline** deklariert sind. Zusätzlich werden hier alle »privaten« Konstanten, Variablen, Prozeduren und Funktionen deklariert, die nur innerhalb der Unit, nicht aber von anderen Programmen oder Units benutzt werden können.

Die Auflistung der Funktions- und Prozedurköpfe im Interface-Teil entspricht im Prinzip einer **forward**-Deklaration, nur daß der Zusatz **forward** hier nicht explizit aufgeführt wird. Auf im Interface-Teil deklarierte Funktionen und Prozeduren kann deshalb im Implementations-Teil ohne Rücksicht auf ihre Reihenfolge im Quelltext Bezug genommen werden.

Die Funktions- und Prozedurblocke, deren Köpfe im Interface-Teil deklariert sind, werden im Implementations-Teil nur durch die Angabe ihres Bezeichners eingeleitet.

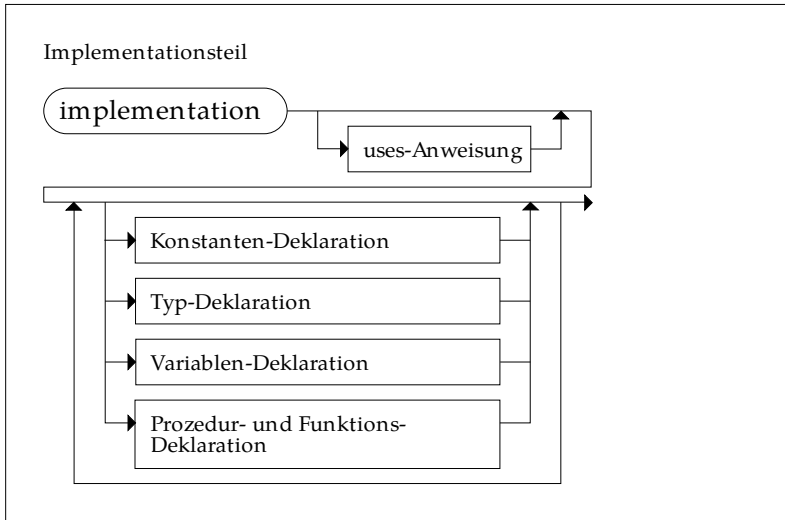


Abbildung 9.7 Syntax des Implementations-Teils einer Unit

Hinweis: Im Gegensatz zu **forward** erlaubt Turbo Pascal hier eine Wiederholung der formalen Parameter-Konstrukte wie das folgende sind also möglich:

```
unit Test;
interface
  procedure P1 (x,y: Integer);
  function F1 (z: Integer): Boolean;
implementation

  procedure P1 (x,y: Integer);
  begin
    ....
  end;

  function F1 (z: Integer): Boolean;
  begin
    ....
  end;
```

Die formalen Parameterlisten innerhalb des Implementationsteils müßten im Standard-Pascal als Kommentare »versteckt« werden. Turbo Pascal nimmt lediglich eine Prüfung auf Übereinstimmung vor (und meldet einen Fehler, wenn angegebene Parameterlisten zwischen **interface** und **implementation** differieren). Durch diese Lockerung der Vorschriften können Units wesentlich übersichtlicher geschrieben werden.

Der Initialisierungs-Teil

ist der letzte Bestandteil einer Unit und stellt das »Hauptprogramm« dar. Er besteht entweder aus dem reservierten Wort **end** (nämlich dann, wenn kein ausführbares »Hauptprogramm« vorhanden ist), oder aus einem mit **begin** eingeleiteten Block, der Anweisungen zur Initialisierung enthalten kann und durch das abschließende **end** der Unit beendet wird.

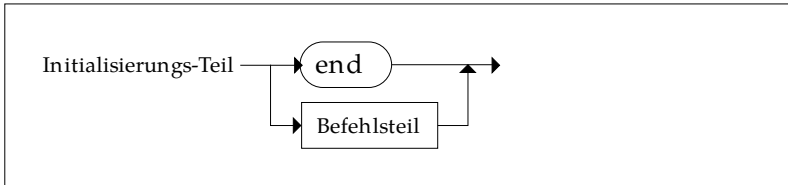


Abbildung 9.8 Syntax des Initialisierungs-Teils einer Unit

Die »Hauptprogramme« der Units, die ein Programm benutzt, werden beim Start des Programms in der Reihenfolge ausgeführt, in der die Unitnamen in der **uses**-Anweisung dieses Programms erscheinen.

Units, die andere Units voraussetzen

Eine **uses**-Anweisung am Anfang einer Unit kann andere Units angeben, die von dieser Unit benutzt bzw. vorausgesetzt werden. Je nach Art der Verkettung ist diese Benutzung direkt oder indirekt:

unit Unit1;	unit Unit2;	program Hauptprogramm;
interface	interface	uses Unit2;
const c = 1;	uses Unit1;	const a = b;
implementation	const b = c;	begin
const d = 2;	implementation	end.
end	end.	

Hier wird *Unit1* von *Unit2* direkt vorausgesetzt und muß deshalb in *Unit2* mit **uses** aufgeführt werden; analoges gilt für *Hauptprogramm* und *Unit2*. Die Beziehung zwischen *Unit1* und *Hauptprogramm* ist dagegen indirekt – sie wird vom Linker automatisch berücksichtigt.

Nach der Veränderung des Interface-Teils einer Unit müssen alle Units neu kompiliert werden, die diese Unit direkt benutzen. Wenn in unserem Beispiel der Interface-Teil von *Unit1* verändert wird (z.B. in $c = 2$), dann ist eine erneute Kompilierung von *Unit2* erforderlich.

Veränderungen von Units, die sich rein auf den Implementations- und/oder Initialisierungsteil beschränken, können dagegen isoliert durchgeführt werden. Im gegebenen Beispiel könnte der Implementations-Teil von *Unit1* verändert werden (z.B. in $d = 1$), ohne daß danach *Unit2* neu kompiliert werden müßte.

Bei der Compilierung einer Unit berechnet Turbo Pascal eine *Unit-Versionsnummer*, die sich als Prüfsumme des Interface-Teils ergibt. Bezogen auf unser Beispiel bedeutet das: bei der Compilierung von *Unit2* wird die aktuelle Versionsnummer von *Unit1* in der compilierten Version von *Unit2* gespeichert. Bei der Compilierung von *Hauptprogramm* vergleicht Turbo Pascal die in *Unit2* gespeicherte Nummer mit dem tatsächlichen Inhalt von *Unit1*. Stimmen die Nummern nicht überein, dann gibt der Compiler (je nach Art der Compilierung) eine Fehlermeldung aus bzw. compiliert *Unit2* neu.

uses-Anweisungen im Implementationsteil

Eine **uses**-Anweisung im Interface-Teil wie

```
uses MyUnit;
```

macht die globalen Bezeichner der Unit *MyUnit* innerhalb der gesamten Unit verfügbar und legt damit gleichzeitig eine strenge Hierarchie fest: Wenn *UnitA* eine weitere Unit *UnitB* mit **uses** aufführt, gibt es keine Möglichkeit, *UnitB* ihrerseits mit **uses** in *UnitA* aufzunehmen.

Daher erlaubt Turbo Pascal zusätzliche **uses**-Anweisungen im Implementationsteil: In dieser Weise aufgenommene Units sind sozusagen »lokal« und kommen sich auch dann nicht in die Quere, wenn sie sich gegenseitig voraussetzen. Das folgende Beispiel demonstriert eine solche Situation, die sich ansonsten nur mit zwei Versionen ein und derselben Routine lösen ließe:

```
program Im_Kreis;
uses Crt, Display; { Die Unit Display enthält die Routine WriteXY }
const Msg = 'Zurück in bewohnte Gebiete';
begin
  ClrScr;
  WriteXY(1,1, 'Obere Linke Ecke des Bildschirms');
  WriteXY(100,100, 'Ziemlich unwahrscheinlich...');
  WriteXY(81 - Length(Msg), 15, Msg);
end.
```

Im_Kreis verwendet eine Routine namens *WriteXY*, die den Text ab der angegebenen Position auf den Bildschirm ausgibt und in der Unit *Display* definiert ist:

```
unit Display;

interface
procedure WriteXY(X,Y: Integer; Msg: string);

implementation
uses
  Crt, Error;

procedure WriteXY(X,Y: Integer; Msg: string);
```

```

begin
  if (X > 0) and (X <= 81-Length(Msg)) and (Y > 0) and (Y <= 25) then
    begin
      GotoXY(X,Y);
      Write(Msg);
    end
    else PrintError('X/Y-Koordinaten außerhalb des Bildschirms!');
  end;
end.

```

Die Routine *WriteXY* der Unit *Display* prüft, ob der als *Msg* übergebene String komplett auf den Bildschirm paßt – falls ja, wird er ausgegeben; falls nein, ruft sie eine Routine namens *PrintError* zur Ausgabe einer Fehlermeldung aus.

PrintError ist in einer Unit namens *Error* definiert, die von *Display* folgerichtig auch mit **uses** aufgeführt wurde:

```

Unit Error;

interface

procedure PrintError(ErrMsg: string);

implementation

uses
  Display;

procedure PrintError(ErrMsg: string);
begin
  WriteXY(1,25, 'Fehler: ' + ErrMsg);
end;

end.

```

Und hier würden die Probleme mit »öffentlichen« **uses**-Anweisungen anfangen: *PrintError* verwendet seinerseits die Routine *WriteXY* zur Ausgabe der Fehlermeldung.

Durch das »Verschieben« beider **uses**-Anweisungen in den Implementations- teil kann Turbo Pascal die Interface-Teile komplett compilieren und damit den Aufruf von *WriteXY* in den Code von *PrintError* einsetzen, weil der Compiler zu diesem Zeitpunkt bereits »weiß«, wie die Deklaration von *WriteXY* aussieht.

Dieser Mechanismus gilt auch für mehrfache gegenseitige Abhängigkeiten (d.h. für Fälle, in denen sich drei oder mehr Units »kreisförmig« aufeinander beziehen). Als Definition formuliert:

Solange die Interface-Teile zweier Units nicht gegenseitig voneinander abhängig sind (z.B. aufgrund gemeinsamer Daten-Deklarationen), können sich Units gegenseitig voraussetzen: die entsprechenden **uses**-Anweisungen werden im Implementationsteil untergebracht; sie müssen dort unmittelbar auf das reservierte Wort **implementation** folgen.

Gemeinsame Datendeklarationen

Die Definition des vorangegangenen Abschnitts läßt gemeinsam benutzte Daten (und Routinen) bewußt aus. Tatsächlich müßten von beiden Units verwendete Variablen, Datentypen oder Routinen im Interface-Teil eines der beiden stehen – womit wir wieder beim alten Problem angelangt wären. Setzen wir beispielsweise einmal voraus, daß die Prozeduren *WriteXY* und *PrintError* unseres Programms *Im_Kreis* zusätzlich die Grenzen eines Ausgabefensters als Parameter erwarten, also:

```
procedure WriteXY(OutputWindow: WindRec; { Fenstergrenzen }
                X,Y: Integer;           { Position im Fenster }
                Msg: string;)          { Text }
```

```
procedure PrintError(OutputWindow: WindRec; { Fenstergrenzen }
                    X,Y: Integer;           { Position im Fenster }
                    Msg: string;)          { Text }
```

Da der Datentyp *WindRec* für beide Routinen zur Verfügung stehen muß, gibt es hier nur eine (aber recht einfache) Lösung: Definieren Sie eine dritte Unit, die nur die entsprechende Typ-Definition enthält:

```
unit WindData;
interface
type
    WindRec = record
        X1,Y1,X2,Y2: Integer; { Fenstergrenzen }
        TColor, BkColor: Integer; { Textattribut }
        Active: Boolean;
    end;
implementation
end.
```

Weil diese Unit keine der beiden anderen voraussetzt, läßt sie sich ohne jedes Problem mit einer **uses**-Anweisung in die Interface-Teile von *Display* und *Error* aufnehmen. (Sowohl *Display* als auch *Error* arbeiten dann mit zwei **uses**-Anweisungen – einer im Interface- und einer im Implementationsteil).

10. Ein- und Ausgabe

Die folgenden Seiten enthalten eine kurze Beschreibung der Prozeduren und Funktionen zur Ein- und Ausgabe (I/O), die in Turbo Pascal eingebaut sind (eine ausführliche Besprechung finden Sie in Kapitel 2 des *Referenzhandbuchs* »Verzeichnis der Routinen«).

Grundlegendes

Variablen des Typs *File* werden als Datei-Variablen bezeichnet, wobei zwischen drei Arten unterschieden wird, nämlich *Textdateien*, *typisierten* und *untypisierten* Dateien. Eine Datei-Variable, die als identisch zum Standardtyp *Text* deklariert worden ist, bezeichnet man als Textdatei. Wird eine Datei-Variable über das Konstrukt **file of** *<Komponententyp>* deklariert, handelt es sich um eine typisierte Datei; eine Deklaration als **file** (ohne Angabe eines Komponententyps) erzeugt eine untypisierte Datei-Variable.

Bevor eine Datei-Variable benutzt werden kann, muß sie einer externen Datei durch den Aufruf der Prozedur *Assign* zugeordnet werden. Bei externen Dateien handelt es sich normalerweise um benannte Disketten-Dateien – *Geräte* wie die Tastatur oder der Bildschirm lassen sich aber ebenfalls als externe Datei behandeln.

Der nächste Schritt nach der Herstellung einer Verbindung zwischen der Datei-Variablen und der externen Datei besteht im *Eröffnen* der Datei, d.h. einigen vorbereitenden Schritten für nachfolgende Ein- und/oder Ausgaben. Eine bereits bestehende Datei kann über die Prozedur *Reset* geöffnet werden, die Prozedur *Rewrite* erzeugt und öffnet dagegen neue Dateien. Mit *Reset* geöffnete Textdateien können nur gelesen, mit *Rewrite* geöffnete Dateien nur beschrieben werden. Typisierte und untypisierte Dateien können sowohl gelesen als auch beschrieben werden, wobei es keine Rolle spielt, ob sie mit *Reset* oder *Rewrite* eröffnet worden sind.

Die vordefinierten Textdatei-Variablen *Input* und *Output* werden mit dem Beginn der Programmausführung automatisch geöffnet. *Input* liest nur Eingaben von der Tastatur, *Output* ist nur für Ausgaben auf den Bildschirm verantwortlich.

Eine Datei besteht aus einer linearen Folge von Komponenten, von denen jede den Komponententyp der Datei hat. Jeder Komponente wird eine *Kompo-*

nenten-Nummer zugeordnet, wobei die erste Komponente der Datei die Nummer 0 erhält.

Auf Dateiinhalte wird normalerweise in sequentieller Folge zugegriffen. Durch das Lesen einer Komponente mit der Standardprozedur *Read* oder das Schreiben mit der Standardprozedur *Write* wird die momentane Position in der Datei auf die nächstfolgende Dateikomponente gesetzt. Auf typisierte und untypisierte Dateien kann dagegen auch nach anderen Kriterien zugegriffen werden: Die Standardprozedur *Seek* setzt die momentane Position in der Datei auf die Komponente der dabei angegebenen Nummer; die Standardprozedur *FilePos* bestimmt die momentane Position in der Datei, über die Standardprozedur *FileSize* kann die momentane Größe (d.h. Komponentenzahl) einer Datei ermittelt werden.

Wenn ein Programm die Bearbeitung einer Datei beendet hat, muß die Datei-Variable mit *Close* wieder *geschlossen* werden. Das Schließen einer Datei-Variablen hat eine Aktualisierung der damit verbundenen externen Datei und eventueller Inhaltsverzeichnisse zur Folge. Danach wird die Verbindung zwischen der Datei-Variablen und der externen Datei gelöst, und die Datei-Variable kann einer anderen externen Datei zugeordnet werden.

Das Laufzeitsystem von Turbo Pascal prüft standardmäßig nach jeder Ein-/Ausgabeoperation auf Fehler und bricht gegebenenfalls mit einer Meldung ab. Diese automatische Überprüfung kann mit den Compiler-Befehlen *{I+}* und *{I-}* an- bzw. abgeschaltet werden. Bei abgeschalteter Überprüfung hat ein Fehler keinen automatischen Abbruch zur Folge, seine Art läßt sich über die Standardfunktion *IOResult* ermitteln.

Standardprozeduren und Funktionen für alle Dateien

Die nachfolgende Tabelle gibt einen kurzen Überblick der eingebauten Funktionen und Prozeduren von Turbo Pascal, die auf Dateien aller drei Arten angewendet werden können.

Assign	ordnet einer Datei-Variablen eine externe Datei über ihren Namen zu
ChDir	wechselt das momentan gesetzte Verzeichnis
Close	schließt eine offene Datei
Eof	prüft, ob das Ende der angegebenen Datei erreicht ist, und liefert einen Wahrheitswert zurück
Erase	löscht eine externe Datei
GetDir	ermittelt das momentan gesetzte Verzeichnis des angegebenen Laufwerks
IOResult	liefert den Status der zuletzt ausgeführten I/O-Operation als Integerwert zurück
MkDir	erzeugt ein Unterverzeichnis
Rename	gibt einer Datei einen neuen Namen
Reset	öffnet eine existierende Datei

Rewrite	erzeugt eine Datei und öffnet sie
RmDir	löscht ein (leeres) Unterverzeichnis

Standard-Prozeduren und Funktionen für Textdateien

Die in diesem Abschnitt beschriebenen Funktionen und Prozeduren sind nur auf Dateien des Typs *Text* anwendbar. Bitte beachten Sie, daß Turbo Pascal zwischen den Datei-Variablentypen *Text* und *File of Char* unterscheidet; im letzteren Fall gilt die Datei als typisiert.

Textdateien werden beim Öffnen auf besondere Art und Weise interpretiert, nämlich als Folge von zeilenweise angeordneten Zeichen, wobei jede Zeile durch ein *Zeilenendezeichen* abgeschlossen ist (␣-Zeichen, ASCII-Code 13, danach eventuell ein Zeilenvorschub (LF), ASCII-Code 10).

Für Textdateien existieren Sonderformen von *Read* und *Write*, die auch das Schreiben und Lesen von Werten ermöglichen, die nicht den Typ *Char* haben. Strings werden als Folge einzelner Zeichen geschrieben und beim Lesen wieder in Strings gespeichert; numerische Werte werden als Folgen numerischer Zeichen geschrieben und beim Lesen zurückverwandelt. Wenn *i* eine Integer-Variable ist, dann liest die Anweisung *Read(f,i)* eine Folge von Ziffern aus der durch *f* bezeichneten Datei, interpretiert sie als dezimalen Integerwert und speichert diesen Wert schließlich in *i*.

Wie bereits erwähnt, deklariert Turbo Pascal die Standard-Dateien *Input* und *Output* automatisch. Beide haben den Typ *Text*, wobei *Input* normalerweise der Tastatur zugeordnet ist und nur gelesen werden kann, *Output* ist normalerweise dem Bildschirm zugeordnet und kann nur beschrieben werden. Die automatische Öffnung von *Input* und *Output* zu Beginn des Programms entspricht den folgenden Anweisungen:

```
var
  Input, Output: Text;

begin
  Assign(Input, '');
  Reset(Input);
  Assign(Output, '');
  Rewrite(Output);
  ...
```

Beide Dateien werden beim Ende des Programms automatisch geschlossen. Das zuvor gezeigte Programmfragment ließe sich also wie folgt abschließen:

```
  Close(Input);
  Close(Output);
end.
```

Hinweis: Bei Verwendung der Unit *Crt* beziehen sich *Input* und *Output* nicht automatisch auf Tastatur und Bildschirm. Kapitel 12 »Standard-Units« enthält

die dazugehörigen Details. Sämtliche der im folgenden beschriebenen Prozeduren und Funktionen arbeiten mit einer externen Textdatei, wenn eine Datei-Variablen angegeben ist – ansonsten beziehen sie sich auf Tastatur bzw. Bildschirm. Folglich hat die Anweisung *Read(x)* dieselbe Bedeutung (und Wirkung) wie *Read(Input, x)*; *Write(x)* und *Write(Output, x)* sind ebenfalls identisch.

Bevor eine dieser Prozeduren und Funktionen auf eine externe Datei angewendet wird, muß diese Datei mit *Assign* zugeordnet und mit *Reset*, *Rewrite* oder *Append* geöffnet werden. Der Versuch, aus einer mit *Rewrite* oder *Append* geöffneten Textdatei etwas zu lesen oder in eine mit *Reset* geöffnete Textdatei etwas hineinzuschreiben, erzeugt einen Laufzeitfehler.

<i>Append</i>	öffnet eine existierende Datei für nachfolgende Schreiboperationen, neue Daten werden »angehängt«
<i>Eoln</i>	prüft, ob in der angegebenen Datei ein Zeilenende erreicht wurde, und liefert einen Wahrheitswert zurück
<i>Flush</i>	erzwingt das physikalische Schreiben eines Ausgabepuffers
<i>Read</i>	liest einen oder mehrere Werte aus der angegebenen Textdatei in die entsprechende Anzahl von Variablen
<i>Readln</i>	hat dieselbe Funktion wie <i>Read</i> , springt aber danach bis zum Anfang der nächsten Zeile in der Textdatei
<i>SeekEof</i>	prüft, ob das Ende der angegebenen Datei erreicht werden kann, und liefert einen Wahrheitswert zurück
<i>SeekEoln</i>	prüft, ob in der angegebenen Datei ein Zeilenende erreicht werden kann, und liefert einen Wahrheitswert zurück
<i>SetTextBuf</i>	ordnet einer Textdatei einen Ein-/Ausgabepuffer zu
<i>Write</i>	schreibt einen oder mehrere Werte in eine Textdatei
<i>Writeln</i>	hat dieselbe Funktion wie <i>Write</i> , schreibt aber danach automatisch ein Zeilenendezeichen

Standard-Prozeduren und Funktionen für untypisierte Dateien

Untypisierte Dateien werden primär für Zugriffe auf niedriger Ebene benutzt, d.h. Zugriffe, bei denen keine Voraussetzungen für Struktur und Inhalt der entsprechenden Disketten-Dateien gemacht werden. Die entsprechenden Datei-Variablen werden durch das reservierte Wort **file** (ohne weitere Zusätze) deklariert:

```
var
  Binaerdatei: file;
```

Bei der Anwendung auf untypisierte Dateien erlauben die Prozeduren *Reset* und *Rewrite* einen weiteren Parameter zur Festlegung der Recordgröße. Die Recordgröße bestimmt, wieviele Bytes bei einem Zugriff gelesen bzw. geschrieben werden; sie wird aus »historischen« Gründen mit 128 angenommen, wenn keine explizite Angabe erfolgt. Vorzuziehen ist allerdings eine Größe von 1, weil

nur dadurch der exakte Umfang der Datei bestimmt werden kann (bei allen anderen Größen wäre ein nur teilweise gefüllter Record am Dateiende möglich).

Abgesehen von *Read*, *Readln*, *Write* und *Writeln* sind alle Standard-Dateiprozeduren und -funktionen von Turbo Pascal auch auf untypisierte Dateien anwendbar. *Read* und *Write* werden durch *BlockRead* und *BlockWrite* ersetzt – zwei Prozeduren, die Block-Datentransfers mit hoher Geschwindigkeit ermöglichen.

BlockRead	liest einen oder mehrere Records aus der angegebenen Datei in eine Variable
BlockWrite	schreibt einen oder mehrere Records aus einer Variablen in die angegebene Datei

Standard-Prozeduren und Funktionen für Dateien

Die folgenden Prozeduren und Funktionen lassen sich auf alle Dateiarnten außer *Text* anwenden:

FilePos	liefert die momentane Position innerhalb einer Datei zurück
FileSize	liefert die Größe der angegebenen Datei zurück
Seek	setzt eine Position innerhalb der angegebenen Datei
Truncate	schneidet eine Datei an der momentanen Position ab

Die Variable FileMode

ist in der Unit *System* deklariert – ihr Wert legt fest, in welchem Modus typisierte und untypisierte Dateien durch *Reset* geöffnet werden. (Dateien des Typs *Text* werden durch *Reset* grundsätzlich für Leseaktionen geöffnet – hier hat *FileMode* keinen Einfluß).

Beim Betrieb unter DOS 2.x kann *FileMode* einem von drei verschiedenen Werten zugewiesen werden:

- 0 Öffnen von untypisierten Dateien nur für Leseaktionen
- 1 nur für Schreibaktionen
- 2 für Lese- und Schreibaktionen gleichzeitig

Die Standardvorgabe für *FileMode* ist der Wert 2 – typisierte und untypisierte Dateien werden von *Reset* also für Lese- und Schreibaktionen gleichzeitig geöffnet.

Die DOS-Versionen 3.x definieren einige weitere Modi für die Bearbeitung von Dateien in Netzen (und wir verweisen an dieser Stelle auf das DOS-Handbuch).

Hinweis: Die Prozedur *Rewrite* öffnet typisierte und untypisierte Dateien grundsätzlich für Lese- und Schreibaktionen gleichzeitig, arbeitet also mit dem

Äquivalent von *FileMode* = 2. (Der aktuelle Wert von *FileMode* wird durch *Rewrite* nicht verändert).

Geräte in Turbo Pascal

Peripheriegeräte werden in Turbo Pascal unter dem Sammelbegriff *Geräte* behandelt; dabei kann es sich um die Tastatur, den Bildschirm, den Drucker oder andere angeschlossene Elektronik handeln. Aus der Sicht des Programmiers werden Geräte als Datei behandelt, wobei dieselben Funktionen und Prozeduren wie bei Diskettendateien zur Anwendung kommen.

Abhängig von der Natur angeschlossener Peripheriegeräte unterscheidet Turbo Pascal zwischen den Geräteklassen »DOS« und »Text«.

DOS-Geräte-einheiten

sind über eine Reihe spezieller (und für diesen Zweck reservierter) Dateinamen implementiert. Die zugrundeliegenden Mechanismen von DOS arbeiten unsichtbar – es macht auch technisch keinen Unterschied, ob sich eine Datei-Variable auf ein DOS-Gerät oder eine Diskettendatei bezieht. Ein Programm wie

```
var
  Lst: Text;

begin
  Assign(Lst, 'LPT1'); Rewrite(Lst);
  Writeln(Lst, 'Eine Ausgabe');
  Close(Lst);
end.
```

gibt die Zeile *Eine Ausgabe* auf dem Drucker aus – obwohl die verwendete Syntax exakt dieselbe ist wie bei der Ausgabe in eine Diskettendatei.

Die durch DOS implementierten Geräte arbeiten in den allermeisten Fällen mit »lesbaren« Ein- und Ausgaben, d.h. werden in Verbindung mit Textdateien benutzt. (Für wenige und spezialisierte Aufgabenstellungen lassen sich auch untypisierte Dateien verwenden). Die nächsten Abschnitte beschreiben die einzelnen DOS-Geräte im Detail, wobei sich – je nach DOS-Version – geringfügige Unterschiede ergeben können.

Die Geräteeinheit CON

bezeichnet das Gerät »Console«, d.h. die Standard-Eingabe (Tastatur) und die Standard-Ausgabe (Bildschirm). Die vorgegebenen Dateien *Input* und *Output* sowie alle Datei-Variablen, denen ein Nullstring (") als Name zugeordnet wird, beziehen sich auf CON, solange Ein- und Ausgaben nicht umgeleitet werden.

Eingaben über CON sind zeilenorientiert und verwenden die im DOS-Hand-

buch beschriebenen Editiermöglichkeiten (`[Backspace]` zum Löschen von Zeichen, Abschluß mit `[a]` usw.). Turbo Pascal liest den von DOS übergebenen Zeilenpuffer zeichenweise – wenn dieser Puffer »ausgelesen« ist, wird eine neue Eingabezeile angefordert. Die Eingabe von `[Ctrl][Z]` erzeugt ein Dateiendezeichen, die Funktion *Eof* liefert nach dieser Eingabe den Wert *True* zurück.

Die Geräteeinheiten LPT1, LPT2 und LPT3

bezeichnen die Schnittstellen zu angeschlossenen Druckern. Solange nur ein Drucker an den Computer angeschlossen ist, wird er normalerweise über LPT1 angesprochen, alternativ kann in diesem Fall die Bezeichnung PRN verwendet werden. Drucker sind reine Ausgabegeräte – die Anwendung von *Reset* auf eine entsprechend zugeordnete Datei-Variable und ein nachfolgender Leseversuch haben das Ergebnis »Dateiende«.

Hinweis: Die Standard-Unit *Printer* deklariert eine Textdatei-Variable namens *Lst*, die LPT1 zugeordnet ist. Die einfachste (und portabelste) Möglichkeit, Ausgaben zu einem Drucker zu senden, besteht in der Aufnahme dieser Unit und den Anweisungen *Write(Lst, ...)* bzw. *WriteLn(Lst, ...)*.

Die Geräteeinheiten COM1 und COM2

bezeichnen die seriellen Schnittstellen des Computers. Anstelle von COM1 kann auch die Bezeichnung AUX verwendet werden.

Hinweis: Lesen und Schreiben von bzw. auf die seriellen Schnittstellen geschieht asynchron, so daß die Anwendung von *Read* bzw. *Write* wenig Sinn ergibt.

Die Geräteeinheit NUL

bezeichnet genau das, was ihr Name impliziert – nämlich eine Datei, die sämtliche Ausgaben ignoriert und bei Leseversuchen sofort ein Dateiendezeichen zurückliefert. NUL wird hauptsächlich dann verwendet, wenn keine neue Datei angelegt werden soll, das Programm aber einen Namen für eine Ein- oder Ausgabedatei erwartet.

Text-Geräte-einheiten

werden verwendet, um von DOS nicht unterstützte Geräte zu implementieren oder vorhandene Geräte zu ersetzen. Die in Kapitel 12 besprochene Standard-Unit *Crt* ist ein gutes Beispiel für den zweiten Fall: sie ersetzt das Gerät CON (also Ein- und Ausgaben via Tastatur und Bildschirm), arbeitet aber wesentlich schneller und bietet eine Fülle von Erweiterungen wie Farben und Fenster.

Geräteeinheiten der Klasse »Text« verwenden (im Gegensatz zu »DOS«) keine reservierten Dateinamen – sie arbeiten über *Gerätetreiber* und werden über spezielle Versionen von *Assign* zugeordnet. Die Unit *Crt* arbeitet beispielsweise mit der (selbstdefinierten) Prozedur *AssignCrt*.

Turbo Pascal ermöglicht die Erstellung eigener Gerätetreiber – die Details finden Sie in Kapitel 14 »Interne Details«.

11. Standardprozeduren und -funktionen

Auf den folgenden Seiten geben wir eine tabellarische Zusammenfassung der eingebauten Prozeduren und Funktionen von Turbo Pascal, die nicht bereits im Zusammenhang mit Ein- und Ausgaben beschrieben wurden (siehe Kapitel 10 »Ein- und Ausgabe«). Weitere Funktionen und Prozeduren finden Sie bei der Beschreibung der Standard-Units im nächsten Kapitel und eine ausführliche Beschreibung sämtlicher verfügbaren Routinen in Kapitel 1 »Verzeichnis der Routinen« des *Referenzhandbuches*.

Standardprozeduren und -funktionen sind in der Unit *System* deklariert, die automatisch als äußerster Block in jedes Programm aufgenommen wird. Aus diesem Grund lassen sich die entsprechenden Bezeichner jederzeit innerhalb des Programms »lokal« redeklieren.

Die Prozeduren Exit, Halt und RunError

Exit	beendet die Ausführung des momentanen Blocks mit sofortiger Wirkung.
Halt	bricht die Ausführung des Programms ab und führt einen Rücksprung zum Betriebssystem aus.
RunError	bricht die Ausführung des Programms durch Simulation eines Laufzeitfehlers ab und führt einen Rücksprung zum Betriebssystem aus. Diese Prozedur ist hauptsächlich zum Austesten von Exit-Routinen gedacht.

Prozeduren und Funktionen zur dynamischen Verwaltung des Speichers

Mit diesen Prozeduren und Funktionen wird der *Heap* verwaltet, d.h. der noch freie Platz im Hauptspeicher bei der Ausführung eines Programms. Eine Besprechung der verwendeten Techniken finden Sie im Abschnitt »Die dynamische Verwaltung des Speichers« in Kapitel 14 »Interne Details«.

Dispose	gibt den Speicherplatz einer dynamischen Variablen wieder frei
FreeMem	gibt einen dynamisch belegten Speicherbereich bestimmter Größe wieder frei
GetMem	belegt einen Speicherbereich der angegebenen Größe für eine dynamische Variable und setzt einen Zeiger auf den Anfang dieses Bereichs
Mark	hält den Zustand des Heaps in einer Zeigervariablen fest
MaxAvail	liefert die Größe des größten freien Heap-Bereichs zurück und ermittelt so, wieviele Bytes momentan maximal für eine dynamische Variable belegt werden können
MemAvail	liefert die Gesamtzahl der freien Bytes auf dem Heap zurück
New	erzeugt eine dynamische Variable und setzt einen Zeiger auf den Anfang des durch diese Variable belegten Bereichs
Release	setzt den Heap auf den mit <i>Mark</i> festgehaltenen Zustand zurück

Transfer-Funktionen

Chr	liefert ein Zeichen mit der angegebenen Ordinalzahl zurück
Ord	liefert die Ordinalzahl des angegebenen (ordinalen) Wertes
Round	konvertiert einen Realwert in einen <i>LongInt</i> und rundet dabei ab oder auf
Trunc	konvertiert einen Realwert in einen <i>LongInt</i> , wobei Nachkommastellen abgeschnitten werden

Hinweis: Die im Standard-Pascal definierten Prozeduren **Pack** und **Unpack** sind in Turbo Pascal nicht verfügbar, weil die entsprechenden Prozesse automatisch ausgeführt werden.

Arithmetische Funktionen

Abs	liefert den absoluten Wert des Arguments zurück
ArcTan	liefert den Arcustangens des Arguments zurück
Cos	liefert den Cosinus des Arguments zurück
Exp	liefert das Ergebnis der Berechnung e^{Argument} zurück, wobei $e = 2.7182\dots$
Frac	liefert die Nachkommastellen des Arguments zurück
Int	liefert den ganzzahligen Anteil des Arguments zurück
Ln	liefert den natürlichen Logarithmus des Arguments zurück
Pi	liefert den Wert <i>Pi</i> (3.141592...) zurück
Sin	liefert den Sinus des Arguments zurück
Sqr	liefert das Quadrat des Arguments zurück
Sqrt	liefert die Quadratwurzel des Arguments zurück

Hinweis: Diese Routinen liefern Werte vom Typ *Real* zurück, wenn ohne Coprozessor bzw. den Emulator gearbeitet wird ({*\$N-*}). Im Modus {*\$N+*} haben die Ergebnisse dagegen den Typ *Extended*:

```
{$N+}
begin
  WriteLn(Pi); { Ergebnis 3.14159265358979E+0000 }
end.

{$N-}
begin
  WriteLn(Pi); { Ergebnis 3.1415926536E+00 }
end.
```

Ordinale Prozeduren und Funktionen

Dec	erniedrigt eine Variable
Inc	erhöht eine Variable
Odd	prüft, ob es sich bei dem Argument um eine ungerade Zahl handelt, und liefert einen Wahrheitswert zurück
Pred	liefert den Vorgänger des Arguments zurück
Succ	liefert den Nachfolger des Arguments zurück

Stringprozeduren und -funktionen

Concat	verbindet eine Folge von Strings und liefert das Ergebnis zurück
Copy	liefert einen Teil eines Strings zurück
Delete	löscht einen Teil eines Strings
Insert	fügt einen Stringteil in einen String ein
Length	liefert die dynamische Länge eines Strings zurück
Pos	sucht eine Zeichenfolge innerhalb eines Strings und liefert ihre Position zurück
Str	konvertiert einen numerischen Wert in einen String
Val	konvertiert einen String in einen numerischen Wert

Zeiger- und Adreß-Funktionen

Addr	liefert die Adresse des angegebenen Objekts zurück
CSeg	liefert den momentanen Wert des CS-Registers zurück
DSeg	liefert den momentanen Wert des DS-Registers zurück

Ofs	liefert die <i>Offset</i> -Adresse des angegebenen Objekts zurück, d.h. die Adresse des Objekts innerhalb eines Segments
Ptr	konvertiert zwei Werte für <i>Segment</i> und <i>Offset</i> in einen Zeigerwert und liefert das Ergebnis zurück
Seg	liefert die Segment-Adresse des angegebenen Objekts zurück
SPtr	liefert den momentanen Wert des SP-Registers zurück
SSeg	liefert den momentanen Wert des SS-Registers zurück

Verschiedenartige Standardprozeduren und -funktionen

FillChar	füllt einen Speicherbereich angegebener Größe byteweise mit einem bestimmten Wert
Hi	liefert das höherwertige Byte des Arguments zurück
Lo	liefert das niederwertige Byte des Arguments zurück
Move	kopiert eine Anzahl von Bytes von einem Speicherbereich in einen anderen
ParamCount	liefert die Anzahl der Kommandozeilen-Parameter zurück, mit denen das Programm gestartet wurde
ParamStr	liefert einen Kommandozeilen-Parameter zurück
Random	liefert eine »Zufallszahl« zurück
Randomize	initialisiert den eingebauten »Zufallszahlen«-Generator
SizeOf	liefert die Größe des angegebenen Objekts in Bytes zurück
Swap	vertauscht höherwertiges und niederwertiges Byte des Arguments miteinander und liefert das Ergebnis zurück
UpCase	konvertiert ein Zeichen in einen Großbuchstaben

12. Standard-Units

In den Kapiteln 10 »Ein- und Ausgabe« und 11 »Standardprozeduren und -funktionen« haben wir die Prozeduren und Funktionen beschrieben, die dem Standard-Sprachumfang von Pascal entsprechen. Sie sind sämtlich in der (grundsätzlich vorhandenen) *Unit System* definiert und deshalb ohne weitere Vorkehrungen verfügbar – im Gegensatz zu den meisten Erweiterungen durch Turbo Pascal, die in separaten Standard-Units deklariert sind. Wir gehen auf den folgenden Seiten hauptsächlich auf die Organisation der Units, Datentypen und Voraussetzungen ein – eine Besprechung der Arbeitsweise sämtlicher Routinen finden Sie in Kapitel 1 »Verzeichnis der Routinen« des *Referenzhandbuches*.

Standard-Units unterscheiden sich in keiner Weise von Units, die Sie als Programmierer auch selber schreiben können. Bevor wir auf die näheren Einzelheiten zu sprechen kommen, erst einmal ein kurzer Abriß:

Crt	erlaubt die vollständige Kontrolle über Ein- und Ausgaben: direkte Zugriffe auf die Tastatur, Setzen der Videomodi, Farben, Fenster und Töne.
Dos	ist die Schnittstelle zum direkten Aufruf von DOS-Funktionen inklusive Datum und Uhrzeit, dem Absuchen von Verzeichnissen und der Ausführung anderer Programme.
Graph	ist ein komplettes Grafik-Paket, das (fast) unabhängig von der verwendeten Hardware arbeitet: unterstützt werden CGA, EGA, VGA, Hercules, PC3270, MCGA, AT&T 6300 und IBM8514.
Graph3	implementiert die Grafik von Turbo Pascal 3.0.
Overlay	verwaltet Overlays und macht so Programme möglich, deren Gesamtumfang die Größe des Hauptspeichers übersteigt.
Printer	vereinfacht Ausgaben auf einen angeschlossenen Drucker.
System	ist die Laufzeitbibliothek von Turbo Pascal und enthält den Standard-Sprachumfang. Diese Unit wird als einziges automatisch in jedes Programm aufgenommen.
Turbo3	sorgt auch bei Spezialfunktionen für Kompatibilität mit der Version 3.0 von Turbo Pascal.

Um eine oder mehrere der Standard-Units in ein Programm aufzunehmen, ist lediglich eine entsprechende **uses**-Anweisung erforderlich. Zum Beispiel:

```
uses Dos, Crt, Graph;
```

Sämtliche Standard-Units außer *Turbo3* und *Graph3* befinden sich normalerweise in der Datei *TURBO.TPL* – sie werden also automatisch zusammen mit Turbo Pascal geladen. Die »Kompatibilitäts-Units« *Turbo3* und *Graph3* kopiert das Installationsprogramm *INSTALL* in ein eigenes Verzeichnis mit dem Namen *TURBO3*, wobei der Menüpunkt *O/D/Unit directories* bzw. die Konfigurationsdatei für die Kommandozeilen-Version des Compilers entsprechend gesetzt werden. (Falls Sie diese Units dennoch direkt in *TURBO.TPL* aufnehmen wollen: Die Beschreibung des Programms *TPUMOVER* in Kapitel 10 »Zusatzprogramme« des *Benutzerhandbuches* enthält eine schrittweise Anleitung dazu).

Durch Standard-Units vorausgesetzte Units

Da die Routinen zur direkten Steuerung des Bildschirms (*ClrScr*, *GotoXY* usw.) seit der Version 3.0 (funktional) unverändert sind, werden sie von den beiden »Kompatibilitäts-Units« *Turbo3* und *Graph3* nicht noch einmal eigens definiert. Um in einem Programm den vollständigen Funktionsumfang der Version 3.0 zu erhalten, müssen Sie deshalb das Unit *Crt* in der **uses**-Anweisung mit aufführen (wobei die Reihenfolge mehr oder weniger beliebig ist):

```
uses Crt, Turbo3, Graph3;
```

Tabelle 12.1 gibt die Zusammenhänge zwischen den Standard-Units von Turbo Pascal wieder:

Tabelle 12.1 Zusammenhänge der Standard-Units

Unit	setzt voraus
System	----
Printer	----
Dos	----
Crt	----
Graph	----
Overlay	----
Turbo3	Crt
Graph3	Crt

(Wir haben in dieser Tabelle absichtlich nicht angegeben, daß jede Unit *System* voraussetzt. *System* wird immer automatisch mit aufgenommen und darf nie in einer **uses**-Anweisung angegeben werden).

Die Unit System

Diese Unit enthält den Standard-Sprachumfang und stellt die Laufzeitbibliothek von Turbo Pascal dar. Hier finden sich sämtliche Unterstützungsroutinen

für die Ein-/Ausgabe auf niedriger Ebene, die Behandlung von Dateien, die Verarbeitung von Strings, mathematische Funktionen, den 8087-Emulator, die dynamische Verwaltung des Speichers und schließlich die Variablen der Overlay-Verwaltung. (Um es noch einmal zu wiederholen: *System* wird automatisch in jedes Programm aufgenommen – eine entsprechende **uses**-Anweisung ist weder notwendig noch erlaubt.)

Hinweis: Die mathematischen Routinen in *System* (*Pi*, *Sqrt*, *Sin* usw.) liefern Werte vom Typ *Real* zurück, wenn ohne Coprozessor bzw. den Emulator gearbeitet wird ({*\$N-*}). Im Modus {*\$N+*} haben die Ergebnisse dagegen den Typ *Extended* (vgl. den Abschnitt »Arithmetische Funktionen« in Kapitel 11 »Standardprozeduren und -funktionen«).

Die in *System* deklarierten Prozeduren und Funktionen sind kurz in den Kapiteln 10 »Ein- und Ausgabe« und 11 »Standardprozeduren und -funktionen« beschrieben, eine ausführliche Besprechung finden Sie in Kapitel 1 »Verzeichnis der Routinen« des *Referenzhandbuchs*.

System deklariert zusätzlich eine Reihe von Variablen und typsierten Konstanten:

const

```
OvrCodeList: Word = 0;      { CSeg-Liste der Overlay-Verwaltung }
OvrHeapSize: Word = 0;      { Größe des Overlay-Puffers }
OvrDebugPtr: Pointer = nil; { !Einhängepunkt! für den Debugger }
OvrHeapOrg: Word = 0;       { Startadresse des Overlay-Puffers }
OvrHeapPtr: Word = 0;       { momentane Spitze des Puffers }
OvrHeapEnd: Word = 0;       { Obergrenze des Puffers }
OvrLoadList: Word = 0;      { Liste der geladenen Segmente }
OvrDosHandle: Word = 0;     { Handle der .OVR-Datei }
OvrEMSHandle: Word = 0;     { dito für OvrInitEMS }
```

Wie die Namen bereits vermuten lassen, werden diese Variablen ausschließlich in Zusammenhang mit Overlays verwendet. Der Overlay-Puffer befindet sich zwischen Stack und Heap des Programms (wobei *OvrHeapOrg* und *OvrHeapEnd* die Start- bzw. Endadresse in Segmenten enthalten; *OvrHeapPtr* zeigt auf die momentane Spitze, d.h. die nächste freie Speicherzelle im Puffer). Die Standardgröße des Puffers entspricht der größten Overlay-Unit des Programms und ist in *OvrHeapSize* (ebenfalls in Segmenten) festgehalten; wenn das Programm ohne Overlays arbeitet, hat diese Variable den Wert 0. Eine allgemeine Erläuterung der zugrundeliegenden Prinzipien finden Sie in Kapitel 3 »Units, Overlays und andere Geheimnisse« des *Benutzerhandbuchs*.

```
HeapOrg: Pointer = nil;      { Start des regulären Heaps }
HeapPtr: Pointer = nil;      { momentane Spitze des Heaps }
HeapEnd: Pointer = nil;      { Ende des regulären Heaps }
HeapError: Pointer = nil;    { für eigene Fehlerbehandlungsroutinen }
FreeList: Pointer = nil;     { Erstes Fragment der FragmentListe }
FreeZero: Word = 0;         { Muß null sein }
```

HeapOrg (»Heap Origin« = Beginn des Heap), *HeapPtr*, *HeapEnd*, *FreeList*, *FreeZero* und *HeapError* werden von den Routinen zur dynamischen Verwaltung des Speichers benutzt – (siehe Kapitel 14 »Interne Details«).

```
ExitProc: Pointer = nil;    { zuletzt eingesetzte Exit-Prozedur }
ExitCode: Integer = 0;     { Exit-Code des Programms }
ErrorAddr: Pointer = nil;  { Adresse eines evtl. Laufzeitfehlers }
```

ExitProc ist eine Zeigervariable, über die eine Prozedur zum Beenden des Programms festgelegt werden kann; *ExitCode* hält den vom Programm an DOS zurückgelieferten Exit-Code fest, *ErrorAddr* die Adresse eines eventuellen Laufzeitfehlers. Auf diese drei Variablen gehen wir im Zusammenhang mit Exit-Prozeduren in Kapitel 14 »Interne Details« noch ausführlicher ein.

```
PrefixSeg: Word = 0;       { PSP-Segmentadresse }
StackLimit: Word = 0;      { Untergrenze des Stacks }
InOutRes: Integer = 0;     { Status für IOResult }
```

PrefixSeg enthält die Segmentadresse des *Programmssegment-Präfix* (PSP) – eines Speicherbereichs, den DOS jedem Programm automatisch voranstellt. Eine detaillierte Beschreibung der in diesem Präfix gespeicherten Daten finden Sie im DOS-Handbuch.

StackLimit legt die niedrigste Adresse innerhalb des Stacksegments fest – wenn der hier angegebene Wert von SP unterschritten wird, bricht die Stackprüfung das Programm mit der Meldung *Stack overflow* ab. Die Standardvorgabe für *StackLimit* ist 0; in einem mit $\{ \$N+, E+ \}$ compilierten Programm reserviert der Emulator die untersten rund 220 Bytes des Stacksegments, falls kein Coprozessor vorhanden ist, und setzt *StackLimit* entsprechend.

InOutRes hält den Wert fest, den der nächste Aufruf von *IOResult* zurückliefert (und erspart dem Programmierer die Definition temporärer Variablen für mehrfache Prüfungen des Ein-/Ausgabestatus).

```
RandSeed: LongInt = 0;    { Startwert des Zufallszahlen-Generators }
FileMode: Byte = 2;       { Standardmodus für das Öffnen von Dateien }
Test8087: Byte = 0;       { Ergebnis der Prüfung auf "Coprozessor" }
```

RandSeed liefert den Startwert des eingebauten »Zufallszahlen«-Generators – durch direkte Zuweisungen können über *Random* erzeugte Zahlenfolgen beliebig oft wiederholt werden. Diese Möglichkeit ist vor allem für das Austesten von Programmen und Verschlüsselungen, Statistiken und Simulationen gedacht. *FileMode* enthält die Standardvorgabe von Turbo Pascal für die Eröffnung von Dateien mit *Reset* (siehe Kapitel 10 »Ein- und Ausgabe«).

Die Variable *Test8087* wird nur gesetzt, wenn das Programm im Modus $\{ \$N+ \}$ compiliert wurde – sie enthält dann das Ergebnis der Prüfung auf »Coprozessor vorhanden« (siehe Kapitel 13 »Der Coprozessor 8087«).

```
var
  Input: Text;             { Standard-Eingabe }
  Output: Text;            { Standard-Ausgabe }
```

Input und *Output* sind die Standard-Dateien zur Ein- und Ausgabe, die durch jedes Turbo Pascal-Programm automatisch geöffnet und der Tastatur bzw. dem Bildschirm zugeordnet werden (siehe Kapitel 10 »Ein- und Ausgabe«).

Das »Hauptprogramm« der Unit *System* belegt unter anderem einige Interrupt-Vektoren des Systems, die erst beim Ende des Programms wieder freige-

geben werden. Vor der Installation der eigenen Behandlungsroutinen speichert *System* die Originalvektoren in einer ganzen Reihe globaler Zeigervariablen:

```
SaveInt00: Pointer; { Vektor $00 - Division durch 0 }
SaveInt02: Pointer; { Vektor $02 - NMI }
SaveInt1B: Pointer; { Vektor $1B - Ctrl-Break }
SaveInt21: Pointer; { Vektor $21 - DOS }
SaveInt23: Pointer; { Vektor $23 - Ctrl-C }
SaveInt24: Pointer; { Vektor $24 - Critical Error }
SaveInt75: Pointer; { Vektor $75 - Gleitkoma-Fehler }
```

Diese sieben Variablen speichern die Originalwerte der Interruptvektoren \$00, \$02, \$1B, \$21, \$23, \$24 und \$75, die durch den Start des Programms auf Routinen der Laufzeitbibliothek gesetzt (und beim Programmende wiederhergestellt) werden.

System setzt den Vektor \$24 auf eine eigene Routine und behandelt »Critical Errors«, auf die DOS normalerweise mit der berüchtigten Meldung *Abort, Retry, Ignore* reagiert, wie ganz normale Ein-/Ausgabefehler: Im Modus *{\$I+}* bricht das Programm mit einem entsprechenden Fehlercode ab, im Modus *{\$I-}* wird lediglich das Ergebnis für den nächsten Aufruf von *IOResult* gesetzt. (Mit dem Aufruf *SetIntVec(\$24, SaveInt24)*; ließe sich der Originalwert des Vektors \$24 wiederherstellen – und damit auch das übliche Reaktionsschema von DOS auf nicht eingelegte Disketten, schadhafte Sektoren usw).

```
SaveInt34: Pointer; { Emulator-Interrupts }
SaveInt35: Pointer;
SaveInt36: Pointer;
SaveInt37: Pointer;
SaveInt38: Pointer;
SaveInt39: Pointer;
SaveInt3A: Pointer;
SaveInt3B: Pointer;
SaveInt3C: Pointer;
SaveInt3D: Pointer;
SaveInt3E: Pointer;
SaveInt3F: Pointer;
```

Diese Vektoren werden nur gespeichert und neu belegt, wenn das Programm im Modus *{\$N+}* kompiliert wurde – über sie wird die Steuerung bzw. Emulation des Coprozessors abgewickelt.

Hinweis: Die Routine *SwapVectors* der Unit *Dos* tauscht die von *System* besetzten Vektoren mit den *SaveIntxx*-Variablen aus und sollte vor und nach jedem Aufruf der Prozedur *Exec* angewendet werden. Auf diese Weise wird sichergestellt, daß ein via *Exec* ausgeführtes Programm nicht versehentlich Routinen der Laufzeitbibliothek des Turbo Pascal-Programms benutzt. Weitere Details finden Sie bei der Besprechung von *SwapVectors* in Kapitel 1 »Verzeichnis der Routinen« des Referenzhandbuchs.

Die Unit Printer

erleichtert Ausgaben auf einen angeschlossenen Drucker: Es deklariert eine Text-Dateivariablen namens *Lst* und ordnet sie der Geräteeinheit LPT1 zu. Die Verwendung von *Printer* hält ein Turbo Pascal-Programm nicht nur portabel, sondern erspart dem Programmierer auch die Deklaration, Zuordnung und Eröffnung einer eigenen Datei am Anfang und schließlich das Schließen dieser Datei am Ende des Programms. Ein kurzes Beispiel für die Verwendung von *Printer* (das wir in Kapitel 10 »Ein- und Ausgabe« schon einmal benutzt haben):

```
program Druckertest;  
uses Printer;  
begin  
  Writeln(Lst, 'Eine Ausgabe');  
end.
```

Die Unit Dos

arbeitet hauptsächlich als Schnittstelle zu den Funktionen von DOS – da Standard-Pascal nichts Vergleichbares kennt, haben wir alle DOS-spezifischen Dinge hier zusammengefaßt. Eine Beschreibung aller Funktionen würde den Rahmen dieses Kapitels sprengen – folglich müssen wir auf das technische Handbuch zu DOS verweisen.

Konstanten, Typen und Variablen

Die folgenden Abschnitte geben einen kurzen Abriß der in der Unit *Dos* deklarierten Bezeichner, wobei nach logischen Funktionsgruppen geordnet wurde.

In Kapitel 1 »Verzeichnis der Routinen« des *Referenzhandbuches* finden Sie eine ausführlichere Besprechung im Zusammenhang mit den jeweiligen Prozeduren und Funktionen, die auf diese Bezeichner zurückgreifen.

Flag-Konstanten

sind für die Auswertung der einzelnen Bits im Flag-Register des Prozessors nach dem Aufruf der Routinen *Intr* und *MsDos* vorgesehen. Die folgenden Konstanten sind zu diesem Zweck in der Unit *Dos* definiert:

```
const  
  FCarry      = $0001;  
  FParity     = $0004;  
  FAuxiliary  = $0010;
```

```

FZero      = $0040;
FSign      = $0080;
FOverflow  = $0800;

```

Wenn wir annehmen, daß *R* ein Record des Typs *Registers* ist, dann liefern die folgenden Tests beide das Ergebnis *True*, wenn das Carry-Flag gesetzt und das Zero-Flag gelöscht ist:

```

R.Flags and FCarry <> 0
R.Flags and FZero = 0

```

Modi der Dateibearbeitung

Die folgenden Konstanten werden von den Routinen zum Öffnen und Schließen von Diskettendateien verwendet. Das Modus-Feld einer Datei-Variablen von Turbo Pascal enthält immer einen der folgenden Werte:

```

const
  fmClosed = $D7B0;
  fmInput  = $D7B1;
  fmOutput = $D7B2;
  fmInOut  = $D7B3;

```

Dateitypen und -records

Die intern von Turbo Pascal verwendeten Record-Typen zur Speicherung von Datei-Variablen sind ebenfalls in der Unit *Dos* definiert. Der Typ *FileRec* wird für typisierte und untypisierte Dateien, der Typ *TextRec* für Textdateien verwendet.

```

type
  FileRec = record { typisierte / untypisierte Dateien }
    Handle:  Word;
    Mode:    Word;
    RecSize: Word;
    Private: array[1..26] of Byte;
    UserData: array[1..16] of Byte;
    Name:    array[0..79] of Char;
  end;
  TextBuf = array[0..127] of Char;
  TextRec = record { Textdateien }
    Handle:  Word;
    Mode:    Word;
    BufSize: Word;
    Private: Word;
    BufPos:  Word;
    BufEnd:  Word;
    BufPtr:  ^TextBuf;
    OpenFunc: Pointer;
    InOutFunc: Pointer;
    FlushFunc: Pointer;
    CloseFunc: Pointer;
    UserData: array[1..16] of Byte;

```

```

Name:      array[0..79] of Char;
Buffer:    TextBuf;
end;

```

Dateiattribut-Konstanten

werden zum Setzen, Testen und Löschen der Attribut-Bits von Dateien über die Prozeduren *GetFAttr*, *SetFAttr*, *FindFirst* und *FindNext* verwendet:

const

```

ReadOnly  = $01;
Hidden    = $02;
SysFile   = $04;
VolumeID  = $08;
Directory = $10;
Archive   = $20;
AnyFile   = $3F;

```

Diese Konstanten sind additiv – ein Aufruf wie

```
FindFirst('*.*', ReadOnly + Directory, S);
```

sucht beispielsweise das momentan gesetzte Verzeichnis nach schreibgeschützten Dateien *und* Unterverzeichnissen ab. Die Konstante *AnyFile* steht für eine Datei mit beliebigen Attributen.

Der Record-Typ Registers

wird von den Prozeduren *Intr* und *MsDos* zur Übergabe der Prozessor-Register verwendet: vor dem Aufruf werden die Register mit den im Record enthaltenen Feldern gesetzt, nach dem Aufruf werden die Ergebnisse in diesem Record gespeichert.

type

```

Registers = record
  case Integer of
    0: (AX, BX, CX, DX, BP, SI, DI, DS, ES, Flags: word);
    1: (AL, AH, BL, BH, CL, CH, DL, DH: Byte);
  end;

```

Wie die Definition zeigt, besteht *Registers* ausschließlich aus varianten Teilen: die 8-Bit-Register des Prozessors belegen denselben Speicherplatz wie ihre »Zusammenfassungen« mit jeweils 16 Bit.

Der Record-Typ DateTime

wird zusammen mit den Prozeduren *PackTime* und *UnpackTime* verwendet, die Datum und Uhrzeit in ein gepacktes Format (4 Byte) konvertieren. Die Prozeduren *GetFTime*, *SetFTime*, *FindFirst* und *FindNext* arbeiten mit Daten dieses gepackten Formats.

type

```
DateTime = record
```

```
Year, Month, Day, Hour, Min, Sec: Integer;
end;
```

Die zulässigen Bereiche für *Month*, *Day*, *Hour* gehen von 1..12, 1..31 und 0..23, für *Min* und *Sec* jeweils von 0..59. Mögliche Werte für das Feld *Year* sind auf den Bereich 1980..2099 beschränkt.

Der Record-Typ SearchRec

wird ausschließlich von den Prozeduren *FindFirst* und *FindNext* verwendet, die ein Verzeichnis nach dem ersten bzw. jeweils nächsten Vorkommen eines Dateieintrags absuchen.

type

```
SearchRec = record
    Fill: array[1..21] of Byte;
    Attr: Byte;
    Time: Longint;
    Size: Longint;
    Name: string[12];
end;
```

Ein Record dieser Art wird von beiden Prozeduren mit den Daten über den gefundenen Dateieintrag gesetzt. Das Feld *Attr* enthält die Attribute der Datei, *Time* das Datum der Erstellung (um hier den »Klartext« zu erhalten, müssen Sie *UnpackTime* anwenden). *Size* enthält die Größe der Datei in Bytes, *Name* den Namen. Das Feld *Fill* wird von DOS zur Speicherung interner Daten (und für die nächste Suche via *FindNext*) verwendet – seinen Inhalt sollte man nie verändern.

Die Variable DosError

wird von den meisten Routinen in der Unit *Dos* für die Übergabe von Fehlercodes verwendet. Ihre Deklaration:

```
var DosError: Integer;
```

Die hier gespeicherten Werte entsprechen direkt den Codes von DOS. 0 steht für »fehlerfreie Ausführung«, andere mögliche Werte und ihre Bedeutung finden Sie in der folgenden Tabelle:

- | | |
|----|--|
| 2 | File not found (Datei nicht gefunden) |
| 3 | Path not found (Suchweg nicht gefunden) |
| 5 | Access denied (Zugriff verweigert) |
| 6 | Invalid handle (Handle nicht definiert/ungültig) |
| 8 | Not enough memory (nicht genügend Platz im Hauptspeicher) |
| 10 | Invalid environment (»Umgebungs«-Parameter ungültig) |
| 11 | Invalid format (ungültiges Befehlsformat) |
| 18 | No more files (keine weiteren Dateieinträge - nur für <i>FindFirst</i> / <i>FindNext</i>) |

Prozeduren und Funktionen

Interrupt-Prozeduren

GetIntVec	ermittelt die Adresse, auf die ein Interrupt-Vektor zeigt
Intr	führt einen Software-Interrupt aus
MsDos	führt einen DOS-Funktionsaufruf aus
SetIntVec	setzt einen Interrupt-Vektor auf eine bestimmte Adresse

Prozeduren für Datum und Uhrzeit

GetDate	ermittelt das momentane Kalenderdatum des Systems
GetFTime	ermittelt Datum und Uhrzeit der letzten Veränderung einer offenen Datei
GetTime	ermittelt die momentan gesetzte Uhrzeit des Systems
PackTime	konvertiert einen Record des Typs <i>DateTime</i> in eine Struktur von 4 Bytes (formal: Typ <i>LongInt</i>) für <i>SetFTime</i>
SetDate	setzt das Kalenderdatum des Systems
SetFTime	setzt Modifikations-Datum und Uhrzeit einer offenen Datei
SetTime	setzt die Uhrzeit des Systems
UnpackTime	konvertiert eine gepackte Struktur von 4 Bytes (formal: Typ <i>LongInt</i>) in einen Record des Typs <i>DateTime</i> . Derartige Strukturen werden von <i>GetFTime</i> , <i>FindFirst</i> und <i>FindNext</i> erzeugt

Statusfunktionen für Disketten- und Festplattenlaufwerke

DiskFree	liefert die Anzahl der freien Bytes auf dem angegebenen Laufwerk zurück
DiskSize	liefert die Gesamtgröße des Datenträgers im angegebenen Laufwerk zurück

Prozeduren zur Bearbeitung von Dateieinträgen

FindFirst	sucht das angegebene bzw. momentan gesetzte Verzeichnis nach dem ersten Eintrag ab, der einem bestimmten Dateinamen und festgelegten Attributen entspricht
FindNext	setzt eine mit <i>FindFirst</i> begonnene Suche fort
GetFAttr	ermittelt die Attribute einer Datei
SetFAttr	setzt die Attribute einer Datei
FSplit	zerlegt einen Dateinamen in seine drei Komponenten »Suchweg«, »Name« und »Suffix«
FExpand	erweitert einen Dateinamen um den Suchweg des momentan gesetzten Verzeichnisse
FSearch	sucht Verzeichnis-Listen nach Dateien ab

Prozeduren und Funktionen für Prozesse

DosExitCode	liefert den Exit-Code eines mit <i>Execute</i> gestarteten Programms zurück
Exec	startet ein Programm, dem zusätzlich Kommandozeilen-Parameter übergeben werden können

Keep	beendet ein Programm und macht es speicherresident
SwapVectors	tauscht die Interruptvektoren des Systems mit den Vektoren der Laufzeitbibliothek und sollte vor bzw. nach jedem Aufruf von <i>Exec</i> verwendet werden

Funktionen zur Ermittlung von Environment-Einträgen

EnvCount	ist das Gegenstück zu <i>ParamCount</i> und liefert die Anzahl der Einträge in der <i>Environment</i> -Tabelle
EnvStr	ist das Gegenstück zu <i>ParamStr</i> und liefert den Eintrag <i>x</i> der <i>Environment</i> -Tabelle
GetEnv	sucht die <i>Environment</i> -Tabelle nicht über Eintragsnummern, sondern über Variablennamen ab

Funktionen und Prozeduren für Parameter von DOS

GetCBreak	ermittelt den momentanen Stand des <i>Break</i> -Flags von DOS
SetCBreak	setzt das <i>Break</i> -Flag von DOS
DosVersion	liefert die Versionsnummer von DOS
GetVerify	ermittelt den momentanen Stand des <i>Verify</i> -Flags von DOS
SetVerify	setzt das <i>Verify</i> -Flag von DOS

Die Unit Crt

ersetzt und erweitert das DOS-Gerät CON: Durch die Aufnahme von *Crt* erhalten Sie die vollständige Kontrolle über Ein- und Ausgaben wie z.B. die Möglichkeit der direkten Tastaturabfrage, das Erzeugen von Farben, Fenstern, Tönen und das Setzen eigener Videomodi. *Crt* arbeitet sehr maschinennah und ist nur auf den Computern der IBM PC-Serie (PC, XT, AT), wirklich »Kompatiblen« und der Modellreihe PS/2 verwendbar. Einer der Hauptvorteile von *Crt* liegt in der hohen Geschwindigkeit, mit der Bildschirm-Operationen durchgeführt werden. Programme, die ohne diese Unit arbeiten, senden ihre Ausgaben zeichenweise über DOS-Funktionsaufrufe, womit ein großer »Überbau« verbunden ist. *Crt* führt je nach Konfiguration entweder direkte BIOS-Aufrufe aus – oder schreibt gleich direkt in den Bildspeicher hinein.

Die Dateien Input und Output

werden durch den Initialisierungs-Teil der Unit *Crt* nicht mehr dem Standardgerät CON zugeordnet, sondern einem speziellen Treiberprogramm innerhalb von *Crt* selbst. Der Ablauf zu Beginn eines Programms entspricht nach der Aufnahme von *Crt* den folgenden Anweisungen:

```
AssignCrt(Input); Reset(Input);
AssignCrt(Output); Rewrite(Output);
```

Daraus folgt: eine Umleitung der Ein- und Ausgabe ist nur dann möglich, wenn die Datei-Variablen *Input* und *Output* innerhalb des Programms explizit wieder dem DOS-Gerät zugeordnet werden, also:

```
Assign(Input, ''); Reset(Input);
Assign(Output, ''); Rewrite(Output);
```

Durch folgende Aufrufe von *AssignCrt* lassen sich die Standard-Dateien jederzeit wieder den Routinen von *Crt* zuordnen.

Fenster

Die Unit *Crt* unterstützt eine einfache, aber äußerst wirkungsvolle Form von Fenstern: über die Prozedur *Window* läßt sich ein »Rahmen« an beliebiger Stelle des Bildschirms festlegen. Schreibaktionen in ein derartiges Fenster laufen exakt so ab, als ob der gesamte Bildschirm verwendet würde: innerhalb des Fensters lassen sich Zeilen löschen oder einfügen, bei Erreichen des rechten Fensterandes wird der Cursor automatisch in die nächste Zeile gesetzt, bei Erreichen des unteren Fensterrandes rollt der Fensterinhalt nach oben usw. Die Teile des Bildschirms, die sich außerhalb des Fensters befinden, bleiben von Aktionen innerhalb des Fensters unberührt.

Alle Bildschirm-Koordinaten außer den Parametern der Prozedur *Window* selbst sind relativ zur linken oberen Ecke des Fensters. Eine Positionierung auf den Punkt (1,1) setzt den Cursor also immer in diese Ecke – unabhängig davon, wo sich das Fenster physikalisch auf dem Bildschirm befindet. Als Standardvorgabe verwendet *Crt* den gesamten Bildschirm als aktives Fenster.

Hinweis: Die Routinen von *Crt* unterstützen den Textmodus mit 43 Zeilen auf einer EGA- bzw. 50 Zeilen auf einer VGA-Karte (siehe den Abschnitt über die Routine *TextMode* in Kapitel 1 »Verzeichnis der Routinen« des *Referenzhandbuchs*).

Spezialzeichen

Bei Schreibaktionen über *Output* oder in eine Datei, die *Crt* zugeordnet worden ist, werden die folgenden Zeichen auf spezielle Weise interpretiert:

Zeichen	Wirkung
#7 (BEL)	erzeugt einen Piepton im eingebauten Lautsprecher
#8 (BS)	bewegt den Cursor um eine Position nach links. Wenn er sich bereits am linken Fensterrand befindet, passiert gar nichts
#10 (LF)	bewegt den Cursor um eine Zeile nach unten, ohne seine Spaltenposition zu verändern. Wenn sich der Cursor bereits in der untersten Zeile des Fensters befindet, wird der gesamte Fensterinhalt entsprechend nach oben gerollt
#13 (CR)	setzt den Cursor auf den linken Fensterrand, ohne seine vertikale Position zu verändern

Alle anderen Zeichen werden ohne Interpretation ausgegeben.

Zeilenweise Eingabe

Leseaktionen von der Standard-Datei *Input* oder einer Textdatei, die *Crt* zugeordnet worden ist, geschehen zeilenweise: eine gelesene Eingabezeile wird im internen Puffer der jeweiligen Datei-Variablen zwischengespeichert und von dort aus je nach Bedarf gelesen. Erst wenn der Inhalt dieses Puffers »ausgelesen« ist und das Programm weitere Eingaben erwartet, folgt eine weitere physikalische Eingabeanforderung.

Bei der zeilenweisen Eingabe über die Tastatur stehen die folgenden Steuer- und Korrekturmöglichkeiten zur Verfügung:

[Backspace]	löscht das zuletzt eingegebene Zeichen und bewegt den Cursor um eine Spalte nach links
[Esc]	löscht sämtliche Zeichen der momentanen Eingabezeile
[a]	schließt die Eingabezeile mit einem Zeilenendezeichen ab
[Ctrl] [S]	hat dieselbe Wirkung wie [Backspace]
[Ctrl] [D]	liest ein Zeichen der vorherigen Eingabezeile bzw. macht eine Löschung mit [Backspace] rückgängig
[Ctrl] [A]	hat dieselbe Wirkung wie [Esc]
[Ctrl] [F]	holt die vorherige Eingabezeile wieder zurück
[Ctrl] [Z]	schließt die Eingabe mit einer Dateiende-Markierung ab

Die Eingabe von [Ctrl] [Z] erzeugt nur dann eine Dateiende-Markierung, wenn die Variable *CheckEof* auf den Wert *True* gesetzt ist (die Standardvorgabe ist *False*, siehe nächster Abschnitt).

Für die direkte Kontrolle der Tastatur und eventueller Eingaben durch das Programm stehen die Funktionen *KeyPressed* und *ReadKey* zur Verfügung, die in Kapitel 1 »Verzeichnis der Routinen« des *Referenzhandbuches* besprochen sind.

Konstanten, Typen und Variablen

Videomodi

Die folgenden Konstanten werden von der Prozedur *TextMode* zum Setzen des Videomodus verwendet:

const

```

BW40    = 0;  { 40x25 S/W, CGA-Karte }
CO40    = 1;  { 40x25 farbig, CGA-Karte }
BW80    = 2;  { 80x25 S/W, CGA-Karte }
CO80    = 3;  { 80x25 farbig, CGA-Karte }
Mono    = 7;  { 80x25 S/W Monochrom-Karte }
Font8x8 = 256; { 80 Zeichen mit 43 bzw. 50
                  Zeilen für EGA, VGA }
C40     = 1;   { entspricht CO40, für 3.0 }
C80     = 3;   { entspricht CO80, für 3.0 }
```

BW40, CO40, BW80 und CO80 stehen für die vier Textmodi, die der *Color Graphics Adapter* (CGA) unterstützt; Mono für den einzigen Modus des Monochrom-Adapters.

Ein Aufruf von *TextMode* mit der globalen Variablen *LastMode* als Parameter setzt den Modus, der vor dem letzten Aufruf von *TextMode* aktiv war, und kann beispielsweise verwendet werden, um von »Grafik« auf »Text« zurückzuschalten; die Addition der Konstanten *Font8x8* zu *BW40..C080* beim Aufruf von *TextMode* setzt 43 Zeilen auf einer EGA- bzw. 50 Zeilen auf einer VGA-Karte.

Text- und Hintergrundfarben

Die folgenden Konstanten werden zusammen mit den Prozeduren *TextColor* und *TextBackground* verwendet:

```
const
{ Für Vorder- und Hintergrund }
  Black      = 0;  { Schwarz }
  Blue       = 1;  { Blau }
  Green      = 2;  { Grün }
  Cyan       = 3;  { Türkis }
  Red        = 4;  { Rot }
  Magenta    = 5;  { Fuchsinrot }
  Brown      = 6;  { Braun }
  LightGray  = 7;  { Hellgrau }

{ Nur für Vordergrundfarben }
  DarkGray   = 8;  { Dunkelgrau }
  LightBlue  = 9;  { Hellblau }
  LightGreen = 10; { Hellgrün }
  LightCyan  = 11; { helles Türkis }
  LightRed   = 12; { Hellrot }
  LightMagenta = 13; { helles Fuchsinrot }
  Yellow     = 14; { Gelb }
  White      = 15; { Weiß }

  Blink      = 128; { blinkend }
```

In den farbigen Textmodi können sämtliche der sechzehn Farben für den Vordergrund verwendet werden, für die Hintergrundfarbe stehen die Farben mit den Nummern von 0 bis 7 zur Verfügung. Durch die Addition des Wertes 128 läßt sich Text zusätzlich blinkend darstellen.

Die Variable CheckBreak

ist standardmäßig auf den Wert *True* gesetzt – wenn der Benutzer Ctrl Break eingibt, wird das Programm bei der nächsten Ein- oder Ausgabe von Text abgebrochen.

```
var
  CheckBreak: Boolean;
```

Ein explizites Setzen dieser Variablen auf *False* unterbindet die Prüfung auf Ctrl Break und einen evtl. Abbruch. (Der Initialisierungs-Teil von *Crt* speichert den Interrupt-Vektor für Ctrl Break in einer globalen Zeigervariablen namens *SaveInt1B* und setzt ihn danach auf eine eigene Routine, über die Eingaben von Ctrl Break abhängig vom Wert der Variablen *CheckBreak* kontrolliert werden).

Die Variable CheckEof

ist standardmäßig auf den Wert *False* gesetzt – wenn der Benutzer ein `[Ctrl] Z` eingibt, wird keine Dateiende-Markierung erzeugt.

var

CheckEof: Boolean;

Ein explizites Setzen dieser Variablen auf den Wert *True* hat zur Folge, daß *Input* nach der Eingabe eines `[Ctrl] Z` bis zu einer erneuten Eröffnung mit *Reset* nur noch Dateiende-Zeichen zurückliefert.

Die Variable CheckSnow

kontrolliert eine spezielle Prüfroutine, die bei direkten Schreibaktionen in den Bildspeicher angewendet wird.

var

CheckSnow: Boolean;

Bei einem *Color Graphics Adapter* älteren Datums erzeugen Zugriffe auf den Bildspeicher, die nicht während des horizontalen Rücklaufs stattfinden, »Schneegestöber« auf dem Bildschirm – kurzzeitige Störungen, die aus einem Zugriffskonflikt zwischen Prozessor und Video-Elektronik herrühren. Monochrom-Adapter, EGAs und VGAs kennen dieses Problem nicht.

CheckSnow ist standardmäßig auf den Wert *True* gesetzt, sobald ein farbiger Textmodus gewählt wird: Schreibaktionen in den Bildspeicher finden nur während des horizontalen Strahlrücklaufes statt.

Wenn Sie einen *Color Graphics Adapter* neueren Datums oder eine EGA-Karte verwenden, können Sie *CheckSnow* zu Beginn des Programms und nach jedem Aufruf von *TextMode* auf den Wert *False* setzen: das Ergebnis ist eine wesentlich höhere Geschwindigkeit bei der Ausgabe von Text.

Hinweis: *CheckSnow* hat keine Wirkung, wenn die Variable *DirectVideo* auf *False* gesetzt ist (siehe nächster Abschnitt).

Die Variable DirectVideo

legt fest, ob *Write* und *Writeln* direkt in den Bildspeicher schreiben oder nicht.

var

DirectVideo: Boolean;

Die Standardvorgabe von *DirectVideo* ist *True*: Schreibaktionen in Dateien, die *Crt* zugeordnet sind, finden als direkte Zugriffe auf den Bildspeicher statt. Wenn Sie das aus irgendeinem Grund verhindern wollen, müssen Sie *DirectVideo* zu Beginn des Programms und nach jedem Aufruf von *TextMode* explizit den Wert *False* zuweisen – *Write* und *Writeln* verwenden dann die (wesentlich langsameren) BIOS-Aufrufe des Computers.

Die Variable LastMode

speichert bei jedem Aufruf der Prozedur *TextMode* den momentanen Video-modus:

var

LastMode: Word;

Da diese Speicherung stattfindet, bevor *TextMode* einen neuen Modus setzt, läßt sich ein »alter« Videomodus (beispielsweise für die Rückschaltung von »Grafik« auf »Text«) recht einfach mit dem Aufruf *TextMode*(*LastMode*) wieder setzen.

Beim Start des Programms wird *LastMode* (durch den Initialisierungs-Teil von *Crt*) auf den momentan aktiven Videomodus gesetzt.

Hinweis: Bei der Umschaltung in einen Modus mit 43 bzw. 50 Textzeilen wird die entsprechende »Zusatzkonstante« *Font8x8* ebenfalls in *LastMode* gespeichert. Der Abschnitt über *TextMode* in Kapitel 1 »Verzeichnis der Routinen« des *Referenzhandbuches* enthält weitere Details.

Die Variable TextAttr

enthält das jeweils gesetzte Attribut für Textzeichen und wird normalerweise durch Aufrufe von *TextColor* bzw. *TextBackground* gesetzt.

var

TextAttr: Byte;

Die Farbe für folgende Zeichenausgaben läßt sich auch über direkte Zuweisungen an *TextAttr* festlegen, wobei die einzelnen Bits innerhalb dieser Variablen folgendermaßen angeordnet sind:



Abbildung 12.1 Bit-Codierung in *TextAttr*

Die vier mit *vvvv* bezeichneten Bits stehen für die Vordergrundfarbe (Werte von 0 bis 15), die mit *hhh* bezeichneten Bits für die Hintergrundfarbe (Werte von 0 bis 7); *B* steht für das Attribut »Blinken«. Wenn Sie die vordefinierten Farbkonstanten verwenden, müssen Sie die Hintergrundfarbe mit dem Wert 16 multiplizieren, um die korrekte Bit-Position zu erreichen. Ein Beispiel:

```
TextAttr := Yellow + Blue * 16 + Blink;
```

Nach dieser Zuweisung werden gelbe blinkende Zeichen auf blauem Hintergrund ausgegeben.

Die Variablen WindMin und WindMax

speichern die (absoluten) Koordinaten des momentan aktiven Fensters und werden normalerweise über Aufrufe der Prozedur *Window* gesetzt.

var

WindMin, WindMax: Word;

WindMin legt die obere linke, *WindMax* die untere rechte Ecke fest, wobei die

X-Koordinate jeweils im niederwertigen Byte und die Y-Koordinate im höherwertigen Byte gespeichert wird. *Lo(WindMin)* liefert also den linken Fensterrand zurück, *Hi(WindMax)* den unteren Rand.

Bei der Zählung ergibt sich ein logisches Problem: die BIOS-Routinen definieren die obere linke Ecke des Bildschirms mit den Koordinaten (0,0), die auch in dieser Form in *WindMin* gespeichert sind – *Window* und *GotoXY* beginnen dagegen mit den Koordinaten (1,1).

Prozeduren und Funktionen

AssignCrt	ordnet dem Bildschirm eine Textdatei-Variable zu
ClrEol	löscht sämtliche Zeichen ab der Position des Cursors bis zum rechten Fensterrand, ohne den Cursor zu bewegen
ClrScr	löscht den gesamten Fensterinhalt und setzt den Cursor in die obere linke Ecke des Fensters
Delay	hält die Programmausführung für die angegebene Anzahl von Millisekunden an und arbeitet unabhängig vom verwendeten Computermodell
DelLine	löscht die Zeile, auf der sich der Cursor befindet, und rollt die darunterliegenden Zeilen des Fensters um eine Zeile aufwärts. Die dadurch freiwerdende unterste Zeile des Fensters wird gelöscht
GotoXY	setzt den Cursor auf die angegebene Spalten- und Zeilenposition. Die Zählung beginnt mit (1,1) und ist relativ zur oberen linken Ecke des Fensters
HighVideo	setzt die Zeichenfarbe auf »hervorgehoben«
InsLine	fügt eine Leerzeile an der Position des Cursors ein. Die folgenden Zeilen des Fensters werden entsprechend abwärts gerollt, die unterste Zeile verschwindet vom Bildschirm
KeyPressed	prüft die Tastatur und liefert <i>True</i> zurück, wenn eine Taste gedrückt wurde, ansonsten ist das Ergebnis <i>False</i>
LowVideo	setzt die Zeichenfarbe auf »halbe Intensität«
NormVideo	setzt die Zeichenfarbe auf »normal«
NoSound	schaltet den eingebauten Lautsprecher ab
ReadKey	liest ein Zeichen von der Tastatur, ohne es automatisch auf den Bildschirm auszugeben
Sound	erzeugt einen Ton im eingebauten Lautsprecher
TextBackground	setzt die Hintergrundfarbe für folgende Textausgaben
TextColor	setzt die Zeichenfarbe für folgende Textausgaben
TextMode	setzt einen (Text-)Videomodus
WhereX	liefert die Spaltenposition (X) des Cursors relativ zur linken Fensterkante zurück. Die Zählung beginnt mit 1
WhereY	liefert die Zeilenposition (Y) des Cursors relativ zur oberen Fensterkante zurück. Die Zählung beginnt mit 1
Window	definiert ein Textfenster auf dem Bildschirm

Die Unit Graph

enthält ein komplettes Grafikpaket mit mehr als 70 Routinen – von bit-orientierten Prozeduren wie *GetImage* und *PutImage* bis zu komplexen Funktionen wie *SetViewPort*, *Circle*, *Bar3D* und *DrawPoly*. *Graph* unterstützt nicht nur verschiedene Arten der Flächenfüllung und unterbrochene Linien, sondern auch diverse Schriftarten, die vergrößert, punktweise versetzt sowie horizontal oder vertikal orientiert werden können.

Um diese Möglichkeiten zu nutzen, sind keine zusätzlichen Dateien erforderlich – *Graph* ist vollständig in der Datei *TURBO.TPL* enthalten, also ein Teil von Turbo Pascal selbst. Zur *Ausführung* eines Programms, das mit *Graph* compiliert wurde, muß allerdings zumindest einer der Grafiktreiber (»Borland Graphics Interfaces« = .BGI-Dateien) vorhanden sein, deren Besprechung Sie im nächsten Abschnitt finden. Wenn Ihr Programm von den Vektor-Zeichensätzen Gebrauch macht, ist darüber hinaus die entsprechende Definitionsdatei (Suffix .CHR) erforderlich.

Im Rahmen der Bedingungen der Benutzerlizenz dürfen Sie die zusammen mit Turbo Pascal gelieferten .BGI- und .CHR-Dateien zusammen mit Ihren Programmen ohne weitere Auflagen weitergeben.

Grafiktreiber und Initialisierungen

Zur Unit *Graph* gehören Treiberprogramme für die folgenden Video-Adapter (und hundertprozentig kompatible Nachbauten):

- CGA (Treiber CGA.BGI)
- MCGA (CGA.BGI)
- EGA (EGAVGA.BGI)
- VGA (EGAVGA.BGI)
- Hercules (HERC.BGI)
- AT&T mit 400 Zeilen (ATT400.BGI)
- 3270 PC (PC3270.BGI)
- IBM 8514 (IBM8514.BGI)

Zur Laufzeit des Programms prüft die Routine *InitGraph* die vorhandene Hardware, lädt den entsprechenden Treiber von der Diskette auf den Heap des Programms und schaltet in den Grafikmodus um. Welcher Treiber von *InitGraph* geladen wurde, läßt sich über *GetDriverName* bestimmen, die Funktion *GetModeName* ermittelt den momentan gesetzten Videomodus (d.h. liefert den Namen dieses Modus als String zurück).

Die Funktion *CloseGraph* ist das Gegenstück zu *InitGraph*: sie stellt den ursprünglichen (Text-)Videomodus wieder her und entfernt den Treiber

aus dem Hauptspeicher. Innerhalb des Programms kann über die Funktionen *RestoreCrtMode* und *SetGraphMode* beliebig zwischen »Text« und »Grafik« umgeschaltet werden.

Alle Grafiktreiber außer »PC3270« und »Hercules« unterstützen mehrere Grafikmodi – welche Modi das im einzelnen sind, läßt sich über *GetModeRange* und *GetMaxMode* ermitteln.

Das Grafikpaket unterstützt Systeme mit zwei Video-Adaptern und Monitoren: bei der Initialisierung durch den Aufruf von *InitGraph* wird automatisch geprüft, welcher der installierten Adapter über den gewünschten Videomodus verfügt; beim Beenden eines Grafikprogramms werden auch in diesem Fall die ursprünglich gesetzten Modi wiederhergestellt. Wenn mehrere Video-Adapter eines Systems grafikfähig sind, wählt *InitGraph* automatisch den Adapter aus, der die höhere Bildqualität bietet.

Auch an »exotische« Video-Adapter ist gedacht: *InstallUserDriver* ermöglicht die Installation zusätzlicher Grafiktreiber; mit *InstallUserFont* lassen sich dem Grafikpaket weitere Zeichensätze hinzufügen.

InitGraph lädt normalerweise immer nur einen Grafiktreiber auf den Heap und erwartet, daß dem Programm die entsprechende .BGI-Datei zur Verfügung steht. Über das Zusatzprogramm BINOBJ (siehe Kapitel 10 »Die Zusatzprogramme« des *Benutzerhandbuches*) lassen sich Grafiktreiber direkt in ein Programm einbinden und sind damit ohne Diskettenzugriffe aktivierbar.

Einige technische Details

Der Ablauf von *InitGraph* besteht aus fünf Schritten:

1. Wenn der zu aktivierende Treiber beim Aufruf nicht explizit festgelegt wurde: Prüfung der Hardware des Systems via *DetectGraph*.
2. Prüfung, ob der entsprechende Treiber durch einen Aufruf von *RegisterBGI-Driver* »registriert« wurde. (Wenn ja, dann wurde der Treiber als .OBJ-Datei eingebunden, ist also Bestandteil des Programms, und es geht mit Schritt 4 weiter).
3. Suchen des Grafiktreibers (*.BGI-Datei) über den zusammen mit *InitGraph* angegebenen Suchweg, Ermittlung der Größe und Belegen von Speicherplatz auf dem Heap über eine Routine namens *GraphGetMem*. Diese Prozedur besteht in der Standardversion lediglich aus einem Aufruf von *GetMem*, wird für alle dynamischen Belegungen des Grafikpakets verwendet und kann vom Programmierer durch eine eigene Version ersetzt werden.
4. Belegen eines Puffers für Polygone und Flächenfüllungen über *GraphGetMem*. Die Größe dieses Puffers läßt sich vor dem Aufruf von *InitGraph* durch die Prozedur *SetGraphBufSize* festlegen.
5. Aktivierung des Treibers, Umschaltung in den Grafikmodus mit *SetGraphMode* und Zurücksetzen aller Parameter auf ihre Standardvorgaben via *GraphDefaults*.

CloseGraph geht den umgekehrten Weg, d.h. schaltet via *RestoreCrtMode* in den Textmodus zurück und gibt dynamisch belegten Speicherplatz durch den Aufruf einer Funktion namens *GraphFreeMem* wieder frei. *GraphFreeMem* besteht

standardmäßig nur aus einem Aufruf von *FreeMem* und ist (wie *GraphGetMem*) nur als eigene Routine definiert, um dem Programmierer die Kontrolle über dynamische Belegungen des Grafikpakets durch selbstgeschriebene Prozeduren zu ermöglichen.

Der Grafik-Cursor

Graph definiert einen *Grafik-Cursor*, der ähnlich wie der normale Cursor im Textmodus arbeitet und die Position der jeweils nächsten Ausgabe bestimmt – nur daß er nicht sichtbar ist. Im Textmodus läßt eine Anweisung wie

```
Write('ABC');
```

den Cursor unmittelbar rechts neben dem ausgegebenen »C« stehen, d.h. in der nächsten Spalte. Wenn das »C« in der äußersten rechten Spalte des Fensters ausgegeben wurde, dann wird der Cursor in die erste Spalte der folgenden Zeile gesetzt; findet die Ausgabe des »C« in der äußersten rechten Spalte der untersten Zeile statt, dann wird der gesamte Fensterinhalt zusätzlich nach oben gerollt.

Der Grafik-Cursor bleibt immer auf dem Punkt der letzten Ausgabe stehen und wird nicht automatisch auf die jeweils nächste freie Position gesetzt. Eine Anweisungsfolge wie

```
MoveTo(0,0);  
LineTo(20,20);
```

setzt ihn also auf den Punkt mit den Koordinaten (20,20). Ein letzter Unterschied: Der Grafik-Cursor wird durch das Abschneiden von Linien (Clipping) bei der Überschreitung von Fenstergrenzen nicht beeinflusst: er zeigt immer auf den zuletzt *angesprochenen* Punkt – unabhängig davon, ob dieser Punkt nun gezeichnet wurde oder nicht.

Die Prozedur *MoveTo* stellt sozusagen das grafische Äquivalent von *GotoXY* dar; sie setzt den Grafik-Cursor auf einen bestimmten Punkt. Die Position des Grafik-Cursors wird nur durch Anweisungen beeinflusst, die Initialisierungen vornehmen oder direkt mit diesem Cursor arbeiten (*InitGraph*, *MoveTo*, *MoveRel*, *LineTo*, *LineRel*, *OutText*, *SetGraphMode*, *ClearDevice*, *SetViewPort* und *ClearViewPort*).

Das Koordinatensystem

Per Definition wird die obere linke Ecke des Bildschirms als *Ursprung* betrachtet, d.h. als der Punkt mit den Koordinaten (0,0). Die X-Koordinate bezeichnet die Spaltenposition (bestimmt also die horizontale Lage) und wird von links nach rechts gezählt, die Y-Koordinate bezeichnet die Zeilenposition (vertikale Lage) und wird von oben nach unten gezählt. Abbildung 12.2 zeigt einen Bildschirm mit einer Auflösung von 320 * 200 Punkten und definiert die Koordinaten der vier Eckpunkte sowie einen Punkt in der Mitte des Bildschirms:

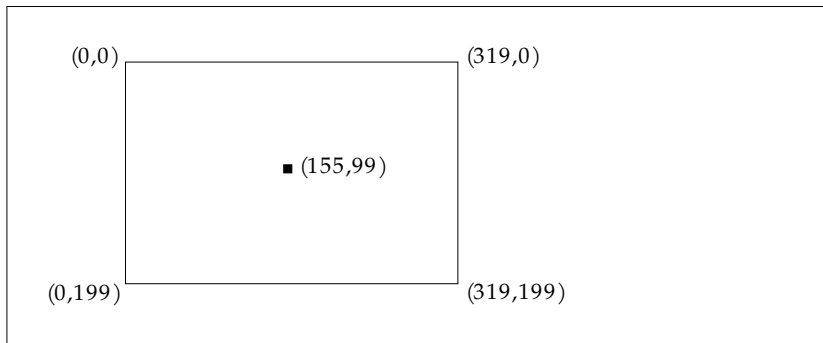


Abbildung 12.2 Eckpunkte in einem System mit 320 * 200 Punkten

Textausgabe

Graph stellt einen bitweise definierten Zeichensatz (8 * 8 Punkte pro Zeichen) sowie mehrere *Vektor-Zeichensätze* zur Verfügung. In einem Vektor-Zeichensatz sind die einzelnen Zeichen nicht punktweise, sondern als Folge relativer Bewegungen (Vektoren) definiert. Der Hauptvorteil dieses etwas höheren Aufwandes: auch bei starken Vergrößerungen werden derartige Zeichen noch in hoher Qualität dargestellt – ganz im Gegensatz zur punktweisen Definition, bei der nur die Punkte entsprechend vergrößert werden können: ab einer gewissen Größe erscheinen Buchstaben hier nur noch als »Klötzchenmuster«.

Alle Vektor-Zeichensätze arbeiten proportional. Mit der Prozedur *SetTextJustify* kann die Ausgabe linksbündig, rechtsbündig oder mittenzentriert justiert werden, für die Auswahl einer Schriftart ist die Prozedur *SetTextStyle*, für beliebige Schriftgrößen die Prozedur *SetUserCharSize* zuständig. Die Ausgabe geschieht entweder über *OutText* oder über *OutTextXY*; mit *GetTextSettings* läßt sich auch nachträglich feststellen, welche Parameter momentan gesetzt sind.

Vektor-Zeichensätze sind jeweils in einer eigenen Datei (Suffix .CHR) definiert, die bei einem entsprechenden Aufruf von *SetTextStyle* verfügbar sein muß.

SetTextStyle lädt immer nur einen Vektor-Zeichensatz auf den Heap und erwartet bei einem Wechsel der Schriftart, daß die entsprechende .CHR-Datei für das Programm verfügbar ist. Über das (in Kapitel 10 »Die Zusatzprogramme« des *Benutzerhandbuchs* besprochene) Zusatzprogramm BINOBJ lassen sich Zeichensätze direkt in ein Programm einbinden und sind damit ohne Diskettenzugriffe aktivierbar.

Objekte und Zeichenstil

Graph enthält sämtliche notwendigen Routinen zum Zeichnen beliebiger Objekte und dem Ausfüllen von Flächen – angefangen von Punkten, Linien, Kreisausschnitten, Rechtecken über Polygone und Balken bis hin zu den bekannten Kuchenstücken und dreidimensionalen Balken. Über *SetLineStyle*

läßt sich festlegen, ob die dazu verwendeten Linien dick oder dünn, durchgehend, gepunktet oder als beliebige Folge von Mustern gezeichnet werden.

Mit den Prozeduren *SetFillStyle*, *SetFloodPattern*, *FillPoly* und *FloodFill* werden Flächen, Bereiche und Polygone entweder mit vorgegebenen Mustern oder frei definierbar ausgefüllt.

Grafikfenster, Bildausschnitte und Grafikseiten

Durch die Prozedur *ViewPort* läßt sich ein rechteckiges Fenster auf einer beliebigen Position des Bildschirms festlegen. Sämtliche Zeichenaktionen arbeiten relativ zum Ursprung (d.h. der oberen linken Ecke) dieses Fensters und finden nur innerhalb der Fenstergrenzen statt, darüber hinausgehende Linien und Figuren werden automatisch abgeschnitten (Clipping).

Wie bereits erwähnt, arbeitet der Grafik-Cursor ebenfalls relativ zum Ursprung des Fensters, wird aber durch den Clipping-Algorithmus nicht beeinflußt; deshalb lassen sich auch Figuren zeichnen, die nur als Ausschnitt innerhalb des Fensters zu sehen sind. Grafik-Fenster lassen sich beliebig oft neu definieren, löschen oder nachträglich bestimmen.

Mit *GetPixel* und *PutPixel* können einzelne Punkte gesetzt bzw. gelesen werden, *GetImage* und *PutImage* lesen bzw. schreiben rechteckige Bereiche des Grafikbildschirms. Sämtliche *BitBlt*-Operationen (Bit Block Transfer) – **xor**, **and**, **or** und **not** – lassen sich nicht nur auf einzelne Punkte, sondern auch auf rechteckige Bildausschnitte anwenden.

Graph enthält Prozeduren zur Auswahl zwischen mehreren Bildschirmseiten für die Erzeugung bewegter Grafiken (soweit der Video-Adapter – Hercules, EGA und VGA – entsprechend RAM zur Verfügung stellt).

Farben und Farb-Paletten

Bei allen farbfähigen Video-Adaptoren legen die für einen Bildpunkt gespeicherten Bits die Farbe nicht direkt fest, sondern dienen als Index in einer *Farb-Palette*, die ihrerseits die tatsächlichen Farb-Bitfolgen enthält:

- Die Anzahl der Einträge einer Farb-Palette (d.h. ihre Größe) ergibt sich aus der Tiefe des Bildspeichers. Bei einer EGA-Karte sind z.B. für jeden Bildpunkt 16 verschiedene Kombinationen möglich, also umfaßt die Palette 16 Einträge.
- Die Größe eines Paletten-Eintrags ist von der Tiefe des Bildspeichers *unabhängig*. Bei einer EGA-Karte kann jeder der 16 Paletten-Einträge Werte im Bereich von 0..63 annehmen – jedem der 16 möglichen Pixelwerte ist also eine von 64 Farben zugeordnet.
- Durch die Veränderung eines Paletten-Eintrags kann das Aussehen des Bildschirms ohne Zugriffe auf den Bildspeicher verändert werden.

Als *Hintergrundfarbe* wird grundsätzlich die Farbe bezeichnet, in der Bildpunkte mit dem Wert 0 auf dem Bildschirm erscheinen.

Die *Zeichenfarbe* legt fest, welchen Wert »gezeichnete« Pixel erhalten, und kann mit einem Aufruf der Form *SetColor(n)* gesetzt werden, wobei *n* für die Nummer eines Paletten-Eintrags steht (CGA: 0..3, EGA: 0..15 usw).

Der Begriff »Farbe« steht also im allgemeinen für den Wert eines Bildpunktes (d.h. die tatsächlich im Bildspeicher eingetragene Bitfolge) – und nicht für die Farbe, in der ein Bildpunkt auf dem Monitor erscheint.

Speicherorganisation und Farbkontrolle für CGA-Karten

Der Color Graphics Adapter (CGA), der AT&T 400 und die MCGA-Karte verwenden im Modus mit 320 * 200 Bildpunkten und 4 Farben eine durchgehende Speicherebene, wobei für jeden Punkt zwei aufeinanderfolgende Bits reserviert sind.

Dem Programmierer stehen vier vordefinierte Farb-Paletten zur Verfügung, bei denen die Einträge 1..3 unveränderlich festgelegt sind; der Eintrag 0 jeder Palette bestimmt die Hintergrundfarbe und kann über die Prozedur *SetBkColor* einen von 16 möglichen Werten zugewiesen bekommen. Die Auswahl einer Palette geschieht über den *Grafikmodus* (CGAC0..CGAC3):

Tabelle 12.2 Vordefinierte Farb-Paletten für CGA-Karten

Name	Wert	Farben		
		1	2	3
CGAC0	0	hellgrün	hellrot	gelb
CGAC1	1	helltürkis	hell-fuchsinrot	weiß
CGAC2	2	grün	rot	braun
CGAC3	3	türkis	fuchsinrot	hellgrau

Für die Farbe des Hintergrundes (Paletten-Eintrag 0) definiert *Graph* die folgenden Konstanten:

Tabelle 12.3 Farbkonstanten von *Graph*

Name	Wert	Bedeutung
Black	0	schwarz
Blue	1	blau
Green	2	grün
Cyan	3	türkis
Red	4	rot
Magenta	5	fuchsinrot
Brown	6	braun
Lightgray	7	hellgrau
Darkgray	8	dunkelgrau
Lightblue	9	hellblau
Lightgreen	10	hellgrün

Name	Wert	Bedeutung
Lightcyan	11	helltürkis
Lightred	12	hellrot
Lightmagenta	13	hell-fuchsinrot
Yellow	14	gelb
White	15	weiß

Hinweis: Die Festlegung der Hintergrundfarbe bei CGA-Karten mit *SetBkColor(color)*, wobei *color* einen Wert von 0..15 hat, ist der einzige Fall, in dem die entsprechende Farbe tatsächlich direkt gesetzt wird – also ohne den »Umweg« über den Eintrag einer Farb-Palette.

CGA-Karten im hochauflösenden Modus

Im »hochauflösenden« Modus des CGA mit 640 * 200 Punkten steht jedes Bit für einen Bildpunkt – folglich sind hier nur noch zwei Farben möglich: Weiß (Bit gesetzt) und Schwarz (Bit gelöscht).

Den Entwicklern von IBM ist in diesem Modus etwas durcheinander geraten: Bildpunkte mit dem Wert 0 (d.h. der Hintergrundfarbe) erscheinen grundsätzlich schwarz, Bildpunkten mit dem Wert 1 läßt sich eine der in Tabelle 12.3 gezeigten Farben zuordnen – und zwar nicht über *SetColor*, sondern mit *SetBkColor*.

Nachdem sich diese Merkwürdigkeit einmal als Standard etabliert hat, halten sich auch die Nachfolger der CGA-Karte daran: dasselbe gilt also für MCGA-Karten in den Modi MCGAMED und MCGAHI sowie für den Video-Adapter des AT&T in den Modi ATT400MED und ATT400HI.

Manipulation von Farb-Paletten

Da die Farb-Paletten bei CGA-Karten fix vorgegeben sind, ist die Funktion *SetAllPaletten* nicht anwendbar; die Funktion *SetPalette*, über die einzelne Einträge einer Farb-Palette gesetzt werden können, läßt sich nur für den Eintrag 0 benutzen (und hat dieselbe Wirkung wie *SetBkColor*).

Speicherorganisation und Farbkontrolle für EGA- bzw. VGA-Karten

EGA- und VGA-Karten arbeiten mit 4 voneinander getrennten Speicher-ebenen; auf jeder Ebene ist für einen Bildpunkt genau ein Bit reserviert, für jeden Bildpunkt sind Werte von 0..15 möglich. Die Farb-Palette enthält 16 Einträge, die jeweils Werte im Bereich von 0..63 annehmen können und frei definierbar sind.

Die Funktion *GetPalette* liefert sämtliche Einträge der Farb-Palette in einem Array zurück; über die Routine *SetPalette* lassen sich die »Hardware-Farben« einzelner Einträge setzen, die Routine *SetAllPalette* setzt sämtliche Einträge mit einem einzigen Aufruf.

Standardmäßig sind den 16 Einträgen der Palette die in der Tabelle 12.3 gezeigten Farben zugeordnet (Eintrag 0 = *Black*, 1 = *Blue*, ... 15 = *White*). Für die dazugehörigen Bitmuster definiert *Graph* ebenfalls symbolische Namen, deren

Auflistung Sie im Abschnitt über *SetAllPalette* in Kapitel 1 »Verzeichnis der Routinen« des *Referenzhandbuches* finden.

Die Prozedur *SetBkColor* erwartet bei EGA- und VGA-Karten einen Wert *n* im Bereich von 1..15, liest den *n*.Paletten-Eintrag und kopiert den gelesenen Wert in den Eintrag 0 der Palette.

... und für den Analog-Adapter IBM8514

Dieser Adapter arbeitet mit 8 Speicherebenen und kann 256 aus 262144 verschiedenen Farben darstellen. Er unterscheidet sich von EGA- und VGA-Karten nicht nur in der Organisation des Bildspeichers und der Art, wie das Videosignal erzeugt wird, sondern leider auch in Bezug auf Manipulationen der Farb-Palette: Jeder der 256 Einträge setzt sich hier aus drei Werten für *Rot*, *Grün* und *Blau* zusammen. *SetPalette*, *GetPalette* und *SetAllPalette* sind deshalb nicht verwendbar – statt dessen stellt Turbo Pascal die Routine *SetRGBPalette* zur Verfügung, mit der sich einzelne Einträge der Farb-Palette setzen lassen.

Damit Turbo Pascal-Programme auch zu Systemen mit dem IBM8514 »kompatibel« (d.h. ohne spezielle Maßnahmen lauffähig) sind, setzt der entsprechende Treiber (IBM8514.BGI) die ersten 16 Einträge der Palette so, daß sie den in Tabelle 12.3 gezeigten »Standardfarben« entsprechen. Der Abschnitt über *SetRGBPalette* in Kapitel 1 »Verzeichnis der Routinen« des *Referenzhandbuches* enthält weitere Details.

Hinweise: Die Routine *FloodFill* läßt sich auf IBM8514-Karten ebenfalls nicht verwenden. Diese (und die zuvor besprochenen) Einschränkungen gelten ebenfalls für VGA-Karten im Modus mit 256 Farben.

**Fehler-
behandlung**

Über die Funktion *GraphResult* lassen sich interne bzw. durch *Graph* ausgelöste Fehler ermitteln. Wie *IOResult* liefert diese Funktion den Ergebniscode der jeweils zuletzt ausgeführten Operation als Integerwert zurück.

Jeder Aufruf von *GraphResult* setzt den Fehlerstatus auf 0 zurück. Um den Fehlerstatus auf mehrere Bedingungen hin zu prüfen, muß er also via *GraphResult* ermittelt und in einer temporären Variablen gespeichert werden, mit der die Prüfungen dann stattfinden. Folgende Codes und Konstanten sind definiert:

Tabelle 12.4 Fehlercodes von *Graph*

Code	Konstante	Bedeutung
0	grOk	fehlerfreie Ausführung
-1	grNoInitGraph	Grafiktreiber (.BGI) nicht installiert – <i>InitGraph</i> muß zuerst aufgerufen werden
-2	grNotDetected	kein grafikfähiger Adapter vorhanden / gewünschter Modus kann nicht gesetzt werden
-3	grFileNotFound	Grafiktreiber-Datei nicht gefunden

Code	Konstante	Bedeutung
-4	grInvalidDriver	Grafiktreiber-Programm defekt (falsche Version, Datei zerstört...)
-5	grNoLoadMem	nicht genug Platz im Hauptspeicher für den Grafiktreiber
-6	grNoScanMem	nicht genug Platz im Hauptspeicher (<i>Bar</i> , <i>FillPoly</i> , <i>PieSlice</i> , <i>Sector</i>)
-7	grNoFloodMem	nicht genug Platz im Hauptspeicher (<i>FloodFill</i>)
-8	grFontNotFound	Zeichensatzdatei nicht gefunden
-9	grNoFontMem	nicht genug Platz im Hauptspeicher für den Zeichensatz
-10	grInvalidMode	Grafikmodus wird vom geladenen Treiber nicht unterstützt
-11	grError	allgemeiner Fehler (hauptsächlich: ungültige Parameter)
-12	grIOError	I/O-Fehler beim Lesen einer .BGI- oder .CHR-Datei
-13	grInvalidFont	ungültige Zeichensatz-Datei
-14	grInvalidFontNum	ungültige Zeichensatznummer

GraphResult bzw. der Fehlerstatus des Grafikpakets wird durch die folgenden Prozeduren und Funktionen gesetzt:

<i>Bar</i>	<i>ImageSize</i>	<i>SetFillPattern</i>
<i>Bar3D</i>	<i>InitGraph</i>	<i>SetFillStyle</i>
<i>ClearViewport</i>	<i>InstallUserDriver</i>	<i>SetGraphBufSize</i>
<i>CloseGraph</i>	<i>InstallUserFont</i>	<i>SetGraphMode</i>
<i>DetectGraph</i>	<i>PieSlice</i>	<i>SetLineStyle</i>
<i>DrawPoly</i>	<i>RegisterBGIDriver</i>	<i>SetPalette</i>
<i>FillPoly</i>	<i>RegisterBGIFont</i>	<i>SetTextJustify</i>
<i>FloodFill</i>	<i>SetAllPalette</i>	<i>SetTextStyle</i>
<i>GetGraphMode</i>		

Einige einfache Beispiele

Das folgende Programm beginnt mit einem Aufruf von *InitGraph*, nachdem zuvor über die Zuweisung *GraphDriver := Detect* festgelegt wurde, daß die Bestimmung des installierten Grafik-Adapters automatisch geschehen soll. *InitGraph* benötigt einen Suchweg für die .BGI-Dateien, solange sich diese Treiberprogramme nicht im momentan gesetzten Verzeichnis befinden: in unserem Beispiel haben wir das Verzeichnis \TP auf dem Laufwerk C: angegeben. Wenn der Computer nicht mit einem grafikfähigen Adapter ausgerüstet ist oder der entsprechende Grafiktreiber nicht geladen werden kann, bricht das Programm mit einer entsprechenden Fehlermeldung ab. Wenn alles so funktioniert, wie es sollte, erhalten Sie einen rechteckigen Rahmen auf dem Bildschirm, in dessen Mitte der Text *Borland Graphics Interface (BGI)* steht.

```

1  program GraphTest1;
2  uses
3    Graph;
4  var
5    GraphDriver : Integer;
6    GraphMode   : Integer;
7    ErrorCode   : Integer;
8  begin
9    GraphDriver := Detect;      { automatische Erkennung }
10   InitGraph(GraphDriver, GraphMode, 'C:\TP');
11   ErrorCode := GraphResult;
12   if ErrorCode <> grOk then      { Fehler? }
13   begin
14     Writeln('Grafik-Fehler: ', GraphErrorMsg(ErrorCode));
15     Writeln('Programm abgebrochen...');
16     Halt(1);
17   end;
18   Rectangle(0, 0, GetMaxX, GetMaxY); { Rahmen max. Größe }
19   SetTextJustify(CenterText, CenterText); { Text zentriert }
20   SetTextStyle(DefaultFont, HorizDir, 3);
21   OutTextXY(GetMaxX div 2, GetMaxY div 2, { Schirmmitte }
22             'Borland Graphics Interface (BGI)');
23   Readln;  { eine Eingabe abwarten }
24   CloseGraph;
25 end.

```

Hinweis: *InitGraph* kann weder den mit 400 Zeilen arbeitenden Grafik-Adapter des AT&T noch die IBM8514-Karte automatisch erkennen. Wenn Sie den Grafik-Adapter des AT&T verwenden, müssen Sie die Zeile 9 des Programms folgendermaßen ersetzen:

```

GraphDriver := ATT400;
GraphMode := ATT400Hi;

```

Dadurch wird *InitGraph* explizit angewiesen, den entsprechenden Treiber (ATT400.BGI) aus dem Verzeichnis C:\TP zu laden und den hochauflösenden Modus (640 * 400 Punkte) zu setzen. Analoges gilt für den IBM8514-Adapter – hier muß die Zeile 9 des Programms folgendermaßen lauten:

```

GraphDriver := IBM8514;
GraphMode := IBM8514Lo;  { oder IBM8514Hi }

```

Ein weiteres Beispielprogramm, das die Umschaltung zwischen Grafik und Text demonstriert:

```

1  program GraphTest2;
2  uses
3    Graph;
4  var
5    GraphDriver : Integer;
6    GraphMode   : Integer;

```

```

7   ErrorCode   : Integer;
8   begin
9   GraphDriver := Detect;           { automatische Erkennung }
10  InitGraph(GraphDriver, GraphMode, 'C:\TP');
11  ErrorCode := GraphResult;
12  if ErrorCode <> grOk then         { Fehler? }
13  begin
14    Writeln('Grafik-Fehler: ', GraphErrorMsg(ErrorCode));
15    Writeln('Programm abgebrochen...');
16    Halt(1);
17  end;
18  OutText('Im Grafikmodus. Weiter mit <RETURN>');
19  Readln;
20  RestoreCRTMode;
21  Write('Im Textmodus. Weiter mit <RETURN>');
22  Readln;
23  SetGraphMode(GraphMode);
24  OutText('Wieder im Grafikmodus. Weiter mit <RETURN>');
25  Readln;
26  CloseGraph;
27 end.

```

Hinweis: Wenn Sie einen AT&T im Modus 640 * 400 oder eine IBM 8514-Karte verwenden, müssen Sie auch hier die Zeile 9 entsprechend ersetzen.

Einige Einzelheiten: Der Aufruf von *SetGraphMode* (in Zeile 23) setzt alle Grafik-Parameter (Farb-Palette, Grafik-Cursor, Vorder- und Hintergrundfarben etc.) auf die Standardwerte zurück. Der Aufruf von *CloseGraph* (in Zeile 26) setzt wieder den Textmodus, der von *InitGraph* zu Beginn des Programms erkannt wurde, und gibt schließlich den vom Treiberprogramm belegten Speicherplatz auf dem Heap frei.

Dynamische Belegung des Speichers

Graph enthält zwei Routinen zur Verwaltung des Heaps: *GraphGetMem* zum Belegen von Platz für Grafik-Treiberprogramme und Zeichensätze sowie *GraphFreeMem* zur Freigabe derselben. Bevor wir weiter darauf eingehen, klären wir erst einmal das Wie und Wo: das Grafikpaket belegt Platz für

- einen Puffer für diverse Zwecke (hauptsächlich: Flächenfüllungen und Polygone), der standardmäßig 4 KByte Größe hat (was sich durch einen Aufruf der Prozedur *SetGraphBufSize* ändern läßt – allerdings nur *vor* dem Start mit *InitGraph*);
- einen Grafiktreiber, der von *InitGraph* als .BGI-Datei von der Diskette geladen wird;
- und den jeweils aktiven Vektor-Zeichensatz, den *SetTextStyle* als .CHR-Datei von der Diskette lädt.

Der Allzweck-Puffer wird in jedem Fall benötigt; das Belegen von Platz für den Grafiktreiber ist dagegen nur notwendig, wenn kein entsprechender Treiber

(nach einer Konvertierung via BINOBJ in eine .OBJ-Datei) direkt in das Programm eingebunden und vor dem Aufruf von *InitGraph* mit *RegisterBGIDriver* »registriert« wurde. Das Belegen von Platz für einen Zeichensatz ist nur notwendig, wenn a) mit Vektor-Zeichensätzen gearbeitet wird und b) die jeweilige .CHR-Datei nicht (ebenfalls nach einer Konvertierung via BINOBJ) direkt in das Programm aufgenommen und vor dem entsprechenden Aufruf von *SetTextStyle* mit *RegisterBGIFont* »registriert« wurde.

Die Deklaration von *GraphGetMem* und *GraphFreeMem*:

```
procedure GraphGetMem(var P: Pointer; Size: Word);
  { Belegung von Speicherplatz für Treiberprogramme }

procedure GraphFreeMem(var P: Pointer; Size: Word);
  { Freigabe belegter Speicherbereiche }
```

Sämtliche Aufrufe dieser beiden Routinen innerhalb von *Graph* geschehen über zwei Zeigervariablen, die global definiert sind:

```
var
  GraphGetMemPtr: Pointer; { Zeiger auf GraphGetMem }
  GraphFreeMemPtr: Pointer; { "" "" GraphFreeMem }
```

Beide Zeiger werden durch den Initialisierungsteil von *Graph* gesetzt und bleiben danach konstant – niemand hält Sie also davon ab, sie auf eigene Routinen zu setzen und so die Verwaltung des »Grafik-Heaps« über selbstgeschriebene Programmteile auszuführen. Voraussetzung dafür ist, daß diese Routinen dieselbe Parameterliste wie *GraphGetMem* bzw. *GraphFreeMem* haben und als *far* codiert sind.

Das folgende Programm demonstriert eine eigene Speicherverwaltung und setzt zusätzlich den Zeiger *ExitProc*, damit *CloseGraph* auch beim Abbruch des Programms aufgerufen wird:

```
program UserHeapManagement;
uses Graph;
var
  GraphDriver,
  GraphMode      : Integer;
  ErrorCode       : Integer; { für den Ergebniscode v. GraphResult }
  PreGraphExitProc : Pointer; { für die originale Exit-Prozedur }

procedure MyGetMem(var P : Pointer; Size : word); far;
  { Belegung von Speicherplatz }
begin
  Write('MyGetMem wurde aufgerufen - weiter mit <RETURN>: '); ReadLn;
  GetMem(P, Size);
end; { MyGetMem }

procedure MyFreeMem(var P : Pointer; Size : word); far;
  { Freigabe von Speicherplatz }
begin
  RestoreCrtMode;
  Write('MyFreeMem wurde aufgerufen - weiter mit <RETURN>: ');
```

```

    ReadLn;
    if P <> Nil then begin { FreeMem darf NIE auf }
        FreeMem(P, Size); { nil-Zeiger angewendet werden! }
        P := Nil;
    end;
end; { MyFreeMem }

procedure MyExitProc; far;
{ Wird bei jeder Art von Programmende aufgerufen }
begin
    ExitProc := PreGraphExitProc; { originale Exit-Routine zurück }
    CloseGraph; { Aufräumen im dynamischen Speicherbereich }

end; { MyExitProc }

begin
    { Installation der eigenen Exit-Routine }
    PreGraphExitProc := ExitProc;
    ExitProc := @MyExitProc;
    GraphGetMemPtr := @MyGetMem; { GraphGetMem ersetzen }
    GraphFreeMemPtr := @MyFreeMem; { dito für GraphFreeMem }

    GraphDriver := Detect;
    InitGraph(GraphDriver, GraphMode, ''); { momentanes Verzeichnis }
    ErrorCode := GraphResult;
    if ErrorCode <> grOk then begin
        WriteLn('Grafik-Fehler: ', GraphErrorMsg(ErrorCode));
        ReadLn;
        Halt(1);
    end;
    Line(0, 0, GetMaxX, GetMaxY);
    OutTextXY(1, 1, 'Ein Text - weiter mit <RETURN>:');
    ReadLn;
end.

```

Konstanten, Typen und Variablen

```

const
    { Ergebniscodes von GraphResult }
    grOk           = 0;
    grNoInitGraph  = -1;
    grNotDetected  = -2;
    grFileNotFound = -3;
    grInvalidDriver = -4;
    grNoLoadMem    = -5;
    grNoScanMem    = -6;
    grNoFloodMem   = -7;
    grFontNotFound = -8;

```

```

grNoFontMem      = -9;
grInvalidMode    = -10;
grError          = -11;  { generischer Fehler }
grIOError        = -12;
grInvalidFont    = -13;
grInvalidFontNum = -14;
{ Grafik-Treiberprogramme }
Detect   = 0;  { automatische Erkennung }
CGA      = 1;
MCGA     = 2;
EGA      = 3;
EGA64    = 4;
EGAMono  = 5;
IBM8514  = 6;
HercMono = 7;
ATT400   = 8;
VGA      = 9;
PC3270   = 10;
CurrentDriver = -128; { aktiver Treiber, für GetModeRange }

{ Mögliche Grafikmodi der einzelnen Treiber }
CGAC0    = 0;  { 320x200, Palette 0: hellgrün,rosa,gelb;
                1 Seite }
CGAC1    = 1;  { 320x200, Palette 1: helltürkis,helles
                Fuchsinrot, weiß; 1 Seite }
CGAC2    = 2;  { 320x200, Palette 2: grün,rot,braun; 1 Seite }
CGAC3    = 3;  { 320x200, Palette 3: cyan, fuchsinrot,hellgrau;
                1 Seite }
CGAHi    = 2;  { 640x200, 1 Seite }
MCGAC0   = 0;  { 320x200, Palette 0: hellgrün,rosa,gelb;
                1 Seite }
MCGAC1   = 1;  { 320x200, Palette 1: helltürkis,helles
                Fuchsinrot,weiß; 1 Seite }
MCGAC2   = 2;  { 320x200, Palette 2: grün,rot,braun; 1 Seite }
MCGAC3   = 3;  { 320x200, Palette 3: cyan,fuchsinrot,hellgrau;
                1 Seite }
MCGAMed  = 4;  { 640x200, 1 Seite }
MCGAHi   = 5;  { 640x480, 2 Farben, 1 Seite }

EGALo    = 0;  { 640x200, 4 Seiten }
EGAHi    = 1;  { 640x350, 16 Farben, 2 Seiten }
EGA64Lo  = 0;  { 640x200, 16 Farben, 1 Seite }
EGA64Hi  = 1;  { 640x350, 4 Farben, 1 Seite }
EGAMonoHi = 3;  { 640x350, 1 Seite mit 64 K, 4 Seiten mit 256 }
HercMonoHi = 0; { 720x348, 2 Seiten }

ATT400C0 = 0;  { 320x200, Palette 0: hellgrün,rosa,gelb;
                1 Seite }
ATT400C1 = 1;  { 320x200, Palette 1: helltürkis,helles

```

```

                                Fuchsinrot,weiß; 1 Seite }
ATT400C2   = 2; { 320x200, Palette 2: grün,rot,braun; 1 Seite }
ATT400C3   = 3; { 320x200, Palette 3: cyan,fuchsinrot,hellgrau;
                                1 Seite }
ATT400Med  = 4; { 640x200, 1 Seite }
ATT400Hi   = 5; { 640x400, 1 Seite }

VGALo      = 0; { 640x200, 16 Farben, 4 Seiten }
VGAMed     = 1; { 640x350, 16 Farben, 2 Seiten }
VGAHi      = 2; { 640x480, 16 Farben, 1 Seite }

PC3270Hi   = 0; { 720x350, 1 Seite }

IBM8514Lo  = 0; { 640 * 480, 256 Farben }
IBM8514Hi  = 1; { 1024 * 768, 256 Farben }

{ Farben für SetPalette und SetAllPalette }
Black      = 0; { schwarz }
Blue       = 1; { blau }
Green      = 2; { grün }
Cyan       = 3; { türkis }
Red        = 4; { rot }
Magenta    = 5; { fuchsin }
Brown      = 6; { braun }
LightGray  = 7; { hellgrau }
DarkGray   = 8; { dunkelgrau }
LightBlue  = 9; { hellblau }
LightGreen = 10; { hellgrün }
LightCyan  = 11; { helles türkis }
LightRed   = 12; { hellrot }
LightMagenta = 13; { helles fuchsinrot }
Yellow     = 14; { gelb }
White      = 15; { weiß }

{ Farben für den IBM-Adapter 8514, die den
Standard-EGA-Farben entsprechen: }
EGABlack   = 0; { dunkle Farben }
EGABlue    = 1;
EGAGreen   = 2;
EGACyan    = 3;
EGARed     = 4;
EGAMagenta = 5;
EGABrown   = 20;
EGALightgray = 7;
EGADarkgray = 56; { helle Farben }
EGALightblue = 57;
EGALightgreen = 58;
EGALightcyan = 59;
EGALightred = 60;
EGALightmagenta = 61;

```

```

EGAYellow      = 62;
EGAWhite       = 63;

{ Linienarten und -breiten für Get/SetLineStyle }
SolidLn       = 0;      { durchgezogen }
DottedLn      = 1;      { gepunktet }
CenterLn      = 2;      { Punkt - Strich - Punkt }
DashedLn      = 3;      { gestrichelt }
UserBitLn     = 4;      { benutzerdefiniert }

NormWidth     = 1;      { normale Breite }
ThickWidth    = 3;      { dick }

{ Set/GetTextStyle }
DefaultFont    = 0;      { 8x8 Bit pixelweise definiert }
TriplexFont   = 1;      { Vektor-Zeichensätze }
SmallFont     = 2;
SansSerifFont = 3;
GothicFont    = 4;

HorizDir      = 0;      { von links nach rechts }
VertDir       = 1;      { von unten nach oben }

UserCharSize  = 0;      { benutzerdefinierte Textgröße }
{ Horizontale/vertikale Justierung für SetTextJustify }
LeftText      = 0;      { linksbündig }
CenterText    = 1;      { zentriert }
RightText     = 2;      { rechtsbündig }

BottomText    = 0;      { unten abschließend }
{ CenterText = 1; }      { siehe oben }
TopText       = 2;      { oben abschließend }

{ Konstanten für das Abschneiden von Linien ("clipping") }
ClipOn       = true;
ClipOff      = false;

{ für Bar3D }
TopOn        = true;
TopOff       = false;

{ Füll-Muster für Get/SetFillStyle }
EmptyFill    = 0;      { Füllen mit der Hintergrundfarbe }
SolidFill    = 1;      { Füllen mit der Vordergrundfarbe }
LineFill     = 2;      { --- }
LtSlashFill  = 3;      { /// }
SlashFill    = 4;      { /// mit dicken Linien }
BkSlashFill  = 5;      { \\ mit dicken Linien }
LtBkSlashFill = 6;      { \\ }
HatchFill    = 7;      { leicht schraffiert }
XHatchFill   = 8;      { stark schraffiert, überkreuzend }
InterleaveFill = 9;      { abwechselnde Linien }

```

```

WideDotFill    = 10; { weit auseinanderstehende Punkte }
CloseDotFill   = 11; { dicht beieinanderstehende Punkte }
UserFill       = 12; { benutzerdefiniert }

{ BitBlt ("Bit Block Transfer") - Möglichkeiten für PutImage
  und SetWriteMode }
CopyPut        = 0;   { MOV } { PutImage / SetWriteMode }
XORPut         = 1;   { XOR } { dito }
OrPut          = 2;   { OR }  { die folgenden drei Möglichkeiten }
AndPut         = 3;   { AND }  { sind nur für PutImage definiert }
NotPut         = 4;   { NOT }

NormalPut = CopyPut; { wg. Kompatibilität zur Version 4.0 }
MaxColors = 15; { für GetPalette, GetDefaultPalette und
                SetAllPalette zur Definition des Typs
                PaletteType }

```

type

```

PaletteType = record { wird von GetPalette, GetDefaultPalette }
    Size : Byte; { und SetAllPalette verwendet }
    Colors : array[0..MaxColors] of ShortInt;
end;

LineSettingsType = record { für GetLineSettings }
    LineStyle : Word;
    Pattern : Word;
    Thickness : Word;
end;

TextSettingsType = record { für GetTextSettings }
    Font : Word;
    Direction : Word;
    CharSize : Word; { 1..10 }
    Horiz : Word;
    Vert : Word;
end;

FillSettingsType = record { für GetFillSettings: }
    Pattern : Word; { Nummer des Füll-Musters }
    Color : Word; { Farbe }
end;

{ für Get/SetFillPattern }
FillPatternType = array[1..8] of Byte; { benutzerdef. Füll-Muster }

PointType = record { vereinfacht die Definition von Parametern }
    X, Y : Integer; { für DrawPoly & FillPoly }
end;

ViewportType = record { für GetViewSettings }
    x1, y1, x2, y2 : Word;

```

```

        Clip                : Boolean;
    end;

    ArcCoordsType = record { für GetArcCoords - enthält die Start- }
        X, Y                : Integer; { und Endpunkte des jeweils letzten }
        Xstart, Ystart,      { mit Arc oder Ellipse gezeichneten }
        Xend, Yend : Integer; { Kreisbogens }
    end;

var
    GraphGetMemPtr : Pointer; { zeigt auf GraphGetMem }
    GraphFreeMemPtr : Pointer; { zeigt auf GraphFreeMem }

```

Prozeduren und Funktionen

Arc	zeichnet einen Kreisbogen von einem Start- zu einem Endwinkel und verwendet eine Koordinate (x,y) als Kreismittelpunkt
Bar	zeichnet einen Balken mit der momentan gesetzten Farbe und dem aktiven Füll-Muster
Bar3D	zeichnet einen dreidimensionalen Balken mit der momentan gesetzten Farbe und dem aktiven Füll-Muster
Circle	zeichnet einen vollständigen Kreis mit gegebenem Radius um einen Mittelpunkt (x,y) herum
ClearDevice	löscht den gesamten Grafikbildschirm (unabhängig von Zeichenfenstern)
ClearViewPort	löscht den Bildschirm im momentan gesetzten Zeichenfenster
CloseGraph	beendet das Grafikpaket
DetectGraph	prüft die vorhandene Hardware und bestimmt, welcher Grafiktreiber geladen und welcher Modus gesetzt werden muß
DrawPoly	zeichnet den Umriß eines Polygons in momentan gesetzter Linienart und -farbe
Ellipse	zeichnet einen elliptischen Kreisausschnitt von einem Start- zu einem Endwinkel um einen Mittelpunkt (x,y) herum
FillEllipse	zeichnet einen ausgefüllten elliptischen Kreis
FillPoly	füllt ein Polygon
FloodFill	füllt einen umschlossenen Bereich mit dem momentan über <i>SetFillStyle</i> bzw. <i>SetFillPattern</i> gesetzten Muster
GetArcCoords	bestimmt die Koordinaten des zuletzt gezeichneten Kreisausschnitts
GetAspectRatio	bestimmt das tatsächliche Darstellungsverhältnis des benutzten Bildschirms
GetBkColor	liefert die momentan gesetzte Hintergrundfarbe zurück
GetColor	liefert die momentan gesetzte Zeichenfarbe zurück
GetDefault-Palette	liefert die Farb-Palette, die bei der Initialisierung des Treibers gesetzt war

GetDriverName	liefert den Namen des aktiven Grafiktreibers als String
GetFillPattern	liefert die einzelnen Bits des momentanen Füll-Musters
GetFillSettings	fragt die über <i>SetFillStyle</i> oder <i>SetFillPattern</i> gesetzten Parameter ab
GetGraphMode	liefert den momentan gesetzten Grafikmodus zurück
GetImage	speichert den Inhalt eines rechteckigen Bildausschnitts bitweise in einer Puffervariablen
GetLineSettings	fragt die durch <i>SetLineStyle</i> gesetzten Parameter ab (Art, Muster und Dicke)
GetMaxColor	liefert die höchste Farbnummer für den momentan aktiven Treiber und Modus
GetMaxMode	liefert die Kennziffer des »höchsten« Modus für den momentan aktiven Treiber
GetMaxX	liefert die größtmögliche X-Koordinate des momentan aktiven Grafikmodus zurück
GetMaxY	liefert die größtmögliche Y-Koordinate des momentan aktiven Grafikmodus zurück
GetModeName	liefert den Namen des aktiven Grafikmodus' als String
GetModeRange	liefert den niedrigst- und höchstmöglichen Grafikmodus für den momentan geladenen Treiber
GetPalette	ermittelt die momentan gesetzte Farb-Palette und ihre Größe
GetPaletteSize	liefert die Anzahl der Paletteneinträge
GetPixel	liefert den Wert des Pixels auf den angegebenen Koordinaten zurück
GetTextSettings	fragt die durch <i>SetTextStyle</i> und <i>SetTextJustify</i> gesetzten Parameter ab (Zeichensatz, Stil und Justierung)
GetViewSettings	ermittelt die Grenzen des momentan gesetzten Grafikfensters und die Bedingungen für das Clipping
GetX	liefert die momentane X-Position des Grafik-Cursors zurück
GetY	liefert die momentane Y-Position des Grafik-Cursors zurück
GraphDefaults	setzt alle Parameter des Grafikpakets auf Standardwerte zurück
GraphErrorMsg	liefert die zum angegebenen Fehlercode gehörige Meldung als String zurück
GraphResult	liefert den momentanen Fehlerstatus des Grafikpakets als Integerwert zurück
ImageSize	liefert die Anzahl von Bytes zurück, die zur Speicherung eines bestimmten Ausschnitts mit <i>GetImage</i> gebraucht werden
InitGraph	initialisiert das Grafikpaket und schaltet in den Grafikmodus um
InstallUserDriver	erlaubt die Installation von Grafiktreibern, die nicht direkt von Borland stammen
InstallUserFont	erlaubt die Installation von Zeichensätzen, die nicht direkt von Borland stammen
Line	zeichnet eine Line von (x1,y1) nach (x2,y2)
LineRel	zeichnet eine Linie relativ zur momentanen Position des Grafik-Cursors

LineTo	zeichnet eine Linie von der momentanen Position des Grafik-Cursors aus zur angegebenen Koordinate
MoveRel	bewegt den Grafik-Cursor von seiner momentanen Position aus um den angegebenen Betrag
MoveTo	setzt den Grafik-Cursor auf den angegebenen Punkt (x,y)
OutText	gibt einen String ab der momentanen Position des Grafik-Cursors aus
OutTextXY	gibt einen String ab der angegebenen Position aus
PieSlice	zeichnet einen Kreisausschnitt, verbindet seine Enden mit dem Mittelpunkt und füllt die entstandene Fläche (Kuchensstück)
PutImage	kopiert den Inhalt einer Puffervariablen in einen rechteckigen Ausschnitt des Bildschirms
PutPixel	zeichnet einen einzelnen Punkt auf den angegebenen Koordinaten
Rectangle	zeichnet ein Rechteck in der momentan gesetzten Farbe und der aktiven Linienart
RegisterBGIDriver	»registriert« einen Grafiktreiber als Teil des Programms
RegisterBGIFont	»registriert« einen Vektor-Zeichensatz als Teil des Programms
RestoreCrtMode	setzt den Videomodus, der beim Aufruf von <i>InitGraph</i> aktiv war
Sector	zeichnet ein ausgefülltes elliptisches »Kuchensstück«
SetActivePage	setzt eine Bildschirmseite für folgende Ausgaben
SetAllPalette	setzt sämtliche Einträge einer Farb-Palette neu
SetAspectRatio	setzt den Korrekturfaktor für optisch runde Kreise direkt
SetBkColor	setzt die Hintergrundfarbe für folgende Zeichenaktionen (und benutzt die aktive Farb-Palette dazu)
SetColor	setzt die Zeichenfarbe für folgende Zeichenaktionen (und benutzt die aktive Farb-Palette dazu)
SetFillPattern	setzt ein benutzerdefiniertes Füll-Muster
SetFillStyle	setzt ein (vordefiniertes) Füll-Muster
SetGraphMode	setzt einen Grafikmodus und löscht den Bildschirm
SetLineStyle	setzt Breite und Muster für nachfolgende Linien
SetPalette	setzt einen Eintrag in einer Farb-Palette
SetRGBPalette	verändert Einträge der Farb-Palette für den Grafik-Adapter 8514 von IBM
SetTextJustify	setzt Parameter für die Justierung von Textausgaben via <i>OutText</i> und <i>OutTextXY</i>
SetTextStyle	setzt einen Zeichensatz, den Stil und den Vergrößerungsfaktor
SetUserCharSize	setzt voneinander unabhängige Zeichenbreiten und -höhen
SetViewPort	setzt ein Zeichenfenster in der Grafik
SetVisualPage	bringt eine von mehreren Grafikseiten auf dem Bildschirm zur Anzeige
SetWriteMode	setzt den Schreibmodus (direkt / XOR) für Linien, die mit <i>DrawPoly</i> , <i>Line</i> , <i>LineRel</i> , <i>LineTo</i> und <i>Rectangle</i> gezeichnet werden

TextHeight liefert die Höhe eines Strings (in Punkten) zurück
 TextWidth liefert die Breite eines Strings (in Punkten) zurück

Eine detaillierte Beschreibung dieser Prozeduren und Funktionen finden Sie in Kapitel 1 »Verzeichnis der Routinen« des *Referenzhandbuchs*.

Die Unit Overlay

implementiert die Overlay-Verwaltung von Turbo Pascal. Da die technische Abwicklung der Prozesse keine Eingriffe von seiten des Programmierers verlangt (oder auch: gestattet) und sich ein Großteil der Variablen in der Unit *System* befindet (wegen der Möglichkeit, auch Overlays schrittweise mit dem integrierten Debugger zu verfolgen), fällt der Interface-Teil von *Overlay* recht kurz aus:

Konstanten, Typen und Variablen

Overlay deklariert die folgenden »öffentlichen« Variablen:

const

```
OvrResult: Integer = 0;
OvrTrapCount: Word = 0;
OvrLoadCount: Word = 0;
OvrFileMode: Byte = 0;
```

OvrResult wird durch den Aufruf jeder Routine von *Overlay* gesetzt und kann sieben verschiedene Werte annehmen.

Für diese hat *Overlay* die folgenden Konstanten deklariert:

const

```
ovrOk = 0;           { "kein Fehler" }
ovrError = -1;       { allg. Fehler }
ovrNotFound = -2;    { Overlay-Datei nicht gefunden }
ovrNoMemory = -3;    { kein Platz zur Vergrößerung des Puffers }
ovrIOError = -4;     { I/O-Fehler beim Zugriff auf die .OVR-Datei }
ovrNoEMSDriver = -5; { EMS-Treiber nicht installiert (OvrInitEMS) }
ovrNoEMSMemory = -6; { nicht genug Platz auf der EMS-Karte ("" ) }
```

Mit Hilfe der Werte von *OvrTrapCount* und *OvrLoadCount* läßt sich die Effizienz des Overlay-Managers überwachen. *OvrFileMode* legt den Zugriffscode fest, der an MS-DOS beim Öffnen der Overlay-Datei übergeben wird.

Außerdem befinden sich die folgenden Deklarationen in der Unit *Overlay*:

type

```
OvrReadFunc = function(OvrSeg: Word): Integer;
```

var

`OvrReadBuf: OvrReadFunc;`

Über die Variable *OvrReadBuf* kann eine eigene Funktion zum Lesen der Overlay-Datei implementiert werden, um beispielsweise DOS-Fehler abzufangen.

Weitere Details zu Overlays finden Sie im Abschnitt »Overlays« in Kapitel 3 »Units, Overlays und andere Geheimnisse« des *Benutzerhandbuchs*.

Prozeduren und Funktionen

OvrInit	startet die Overlay-Verwaltung und ist Voraussetzung für alle anderen Aufrufe von <i>Overlay</i>
OvrInitEMS	kopiert die .OVR-Datei des Programms in den Speicher einer EMS-Karte (soweit vorhanden)
OvrClearBuf	wirft sämtliche momentan geladenen Overlay-Units aus dem Hauptspeicher und erzwingt so eine Reorganisation des Overlay-Puffers
OvrGetBuf	liefert die momentane Größe des Overlay-Puffers zurück
OvrSetBuf	legt eine neue Größe des Overlay-Puffers fest
OvrGetRetry	liefert die momentane Größe des Bewährungsbereichs im Overlay-Puffer zurück
OvrSetRetry	legt eine neue Größe des Bewährungsbereichs fest

Die Unit Turbo3

Die Prozeduren und Funktionen dieser Unit sind ausschließlich zur Herstellung der »Abwärtskompatibilität« mit der Version 3.0 von Turbo Pascal vorgesehen; jede dieser Routinen existiert (meist in verbesserter und erweiterter Form) auch in den anderen Standard-Units. Durch die Verwendung von *Turbo3* wird die Übernahme alter Programme erleichtert – allerdings erschweren Sie sich damit den Zugang zu den neuen Möglichkeiten, die Ihnen seit der Version 4.0 zur Verfügung stehen.

Da Turbo Pascal die Möglichkeit der »qualifizierten« Angabe eines Bezeichners kennt, können Sie auf die neuen Routinen aber auch dann zugreifen, wenn *Turbo3* in das Programm aufgenommen wurde. Nach einer **uses**-Anweisung wie

uses

`Crt, Turbo3;`

aktiviert beispielsweise der Bezeichner *KeyPressed* die entsprechende Routine in *Turbo3*, der Bezeichner *System.KeyPressed* dagegen die Funktion innerhalb der Laufzeitbibliothek der Version 6.0. Kapitel 3 »Units, Overlays und andere Geheimnisse« des *Benutzerhandbuchs* und Kapitel 8 »Prozeduren und Funktio-

nen« in diesem Handbuch setzen sich genauer mit Bezeichnern auseinander, die in mehr als einer Unit deklariert sind.

Hinweis: Die in den folgenden Abschnitten aufgeführten Routinen von *Turbo3* sind *nicht* in Kapitel 1 »Verzeichnis der Routinen« des *Referenzhandbuchs* beschrieben – wir verweisen hier auf das Benutzerhandbuch der Version 3.0.

Der Interface-Teil

deklariert zwei globale Variablen, fünf Funktionen und vier Prozeduren:

```
unit Turbo3;
interface
uses Crt;
var
    Kbd: Text;
    CBreak: Boolean absolute CheckBreak;
procedure AssignKbd(var F: Text);
function MemAvail: Integer;
function MaxAvail: Integer;
function LongFileSize(var F): Real;
function LongFilePos(var F): Real;
procedure LongFileSeek(var F; Pos: Real);
```

Hinweis: Die Routine *AssignKbd* setzt nur das Feld *Mode* der als *F* übergebenen Variablen. (Um es genau zu nehmen, ist *Kbd* in *Turbo3* als typisierte Konstante definiert, bei der die restlichen Felder vorgegeben sind).

Die Variable Kbd

stand in der alten Version für das direkte Lesen der Tastatur mit der Anweisung *Read(Kbd, CharX)* zur Verfügung (wobei *CharX* eine Variable vom Typ *Char* ist). Die Version 6.0 ersetzt diese Prozedur durch die in *Crt* deklarierte Funktion *ReadKey*. Ein Beispiel für die Modifikation eines Programms, das jeweils ein Zeichen von der Tastatur liest und zwischen normalen Tasten und Funktions-tasten (wie *[F1]*, *[F2]*, den Cursortasten usw.) unterscheidet:

```
program TestKbd; { kompatibel zur Version 3.0 }

uses Crt, Turbo3;

var
    ch: Char;

begin
    Read(Kbd, ch);
    if (ch = #27) and KeyPressed then
        begin
            Read(Kbd, ch);
            WriteLn('Spezialtaste: ', ch);
        end
```

```

    else Writeln('Normale Taste: ', ch);
end.

```

Der Gerätetreiber für *Kbd* verwandelt erweiterte Tastaturcodes von *NUL* + *Zeichen* in `[Esc] + Zeichen`. Da `[Esc]` natürlich auch direkt eingegeben worden sein kann, ist in jedem Fall eine Prüfung via *KeyPressed* notwendig, um zwischen einer Spezialtaste und `[Esc]` zu unterscheiden. Noch etwas unerfreulicher: falls der Benutzer schnell genug zwei Zeichen hintereinander eingibt, von denen das erste ein `[Esc]` ist, versteht das Programm diese beiden Eingaben als einen Druck auf eine Spezialtaste.

Die neuere Version:

```

program TestReadKey; { Version 6.0 }
uses
  Crt;
var
  ch: Char;
begin
  ch := ReadKey;
  if (ch = #0) then
    Writeln('Spezialtaste: ', ReadKey);
  else Writeln('Normale Taste: ', ch);
end.

```

Dieses Programm ist nicht nur kürzer, es vermeidet auch mögliche Mehrdeutigkeiten der Interpretation. (Im Gegensatz zu `[Esc]` kann ein Zeichen mit dem ASCII-Code 0 nur über die Spezialtasten erzeugt werden).

Die Variable CBreak

hat dieselbe Funktion wie die Variable *CheckBreak* der Version 6.0 (in der Unit *Crt*) und wird in *Turbo3* einfach auf derselben Adresse neu deklariert. Mit der Zuweisung *CBreak* := *False* läßt sich die Prüfung auf `[Ctrl] [Break]` bei Ein- und Ausgaben über Tastatur und Bildschirm abschalten, *CBreak* := *True* schaltet diese Überprüfung an.

Prozeduren und Funktionen

MemAvail	liefert die Gesamtgröße des freien Speicherbereichs auf dem Heaps in Paragraphen (Einheiten zu je 16 Byte) als Integer zurück
MaxAvail	liefert die Größe des größten zusammenhängenden freien Speicherbereichs auf dem Heaps in Paragraphen als Integer zurück
LongFileSize	liefert die Größe einer Datei im Format <i>Real</i> zurück
LongFilePos	liefert die Position innerhalb einer Datei im Format <i>Real</i> zurück

LongSeek	setzt die Position innerhalb einer Datei auf die angegebene Komponente und erwartet einen Realwert als Argument
HighVideo	setzt die Zeichenfarbe für Textausgaben bei Farbbildschirmen auf »gelb mit schwarzem Hintergrund«, bei monochromen Systemen auf »weiß mit schwarzem Hintergrund«
NormVideo	hat dieselbe Funktion wie <i>HighVideo</i>
LowVideo	setzt die Zeichenfarbe für Textausgaben auf »hellgrau mit schwarzem Hintergrund«

Die Unit Graph3

Diese Unit stellt eine Implementation der Grafikroutinen und der Turtlegrafik von Turbo Pascal 3.0 dar, die in dieser Version allerdings in zwei Dateien unterteilt ist: GRAPH.P (den Deklarationen) und GRAPH.BIN (dem tatsächlichen Maschinencode).

Graph3 faßt GRAPH.P und GRAPH.BIN in einer einzigen Datei zusammen und ist ein direkter Ersatz: die einzig notwendige Änderung für Programme der Version 3.0 besteht im Austausch des Compiler-Befehls `{I GRAPH.P}` gegen die Anweisung `uses Crt, Graph3`.

Hinweis: Die im nächsten Abschnitt aufgeführten Routinen sind *nicht* weiter in Kapitel 1 »Verzeichnis der Routinen« des *Referenzhandbuches* beschrieben – wir verweisen auch hier auf das Handbuch der Version 3.0.

Prozeduren und Funktionen

GraphColorMode	setzt den Modus »320 * 200 Punkte, 4 Farben«
GraphMode	setzt den Modus »320 * 200 Punkte, schwarzweiß«
HiRes	setzt den Modus »640 * 200 Punkte, zweifarbig«
Palette	aktiviert die angegebene Farb-Palette
HiResColor	setzt eine Zeichenfarbe für den Modus <i>HiRes</i>
GraphBackground	setzt die Hintergrundfarbe
GraphWindow	legt ein Zeichenfenster auf dem Bildschirm fest
Plot	zeichnet einen Punkt mit der angegebenen Farbe auf den angegebenen Koordinaten
Draw	zeichnet eine Line zwischen zwei angegebenen Punkten in der angegebenen Farbe
ColorTable	definiert eine Farb-Übersetzungstabelle, über die die Farbe eines Punktes bei nachfolgenden Zeichenaktionen als Verknüpfung mit dem vorherigen Bildinhalt festgelegt wird
Arc	zeichnet ein Kreisbogenstück mit festgelegtem Radius sowie Start- und Endwinkel
Circle	zeichnet einen vollständigen Kreis (entspricht <i>Arc</i> mit der Angabe 0 bzw. 360 für Start- und Endwinkel)

GetPic	kopiert einen rechteckigen Ausschnitt des Bildschirms in eine Puffervariable
PutPic	kopiert den Inhalt einer Puffervariablen in den Bildschirm
GetDotColor	liefert die Farbe eines Punktes zurück
FillScreen	füllt das gesamte aktive Fenster mit einer Farbe
FillShape	füllt eine umschlossene Fläche mit einer Farbe
FillPattern	füllt einen rechteckigen Bereich des Bildschirms mit dem momentan aktiven Muster in der angegebenen Farbe
Pattern	definiert ein Muster mit $8 * 8$ Punkten, das von <i>FillPattern</i> verwendet wird

Turtlegrafik

Back	bewegt den Zeichenstift um den angegebenen Betrag entgegen der momentanen Zeichenrichtung
Forwd	bewegt den Zeichenstift um den angegebenen Betrag in der momentanen Zeichenrichtung
ClearScreen	löscht den Inhalt des Zeichenfensters und setzt den Zeichenstift in die Mitte dieses Fensters
Heading	liefert die momentane Bewegungsrichtung des Zeichenstiftes zurück
HideTurtle	läßt die Schildkröte (den Zeichenstift) vom Bildschirm verschwinden
Home	setzt den Zeichenstift auf seine Anfangsposition zurück
NoWrap	legt fest, daß der Zeichenstift nicht automatisch an der anderen Seite des Zeichenfensters wieder erscheint, wenn er eine Fensterkante überschritten hat
PenDown	setzt den Zeichenstift an, nachfolgende Bewegungen hinterlassen einen Strich
PenUp	hebt den Zeichenstift ab, nachfolgende Bewegungen hinterlassen keine Spuren
SetHeading	legt die Bewegungsrichtung fest
SetPencolor	legt die Zeichenfarbe fest
SetPosition	bewegt den Zeichenstift an die angegebene Position, ohne dabei zu zeichnen
ShowTurtle	läßt die Schildkröte (den Zeichenstift) auf dem Bildschirm erscheinen
TurnLeft	verändert die Bewegungsrichtung um den angegebenen Winkel gegen den Uhrzeigersinn
TurnRight	verändert die Bewegungsrichtung um den angegebenen Winkel im Uhrzeigersinn
TurtleWindow	legt ein Zeichenfenster auf dem Bildschirm fest
TurtleThere	prüft, ob der Zeichenstift sichtbar ist, und liefert ein boolesches Ergebnis zurück
TurtleDelay	legt eine Verzögerung nach jedem Schritt des Zeichenstiftes fest
Wrap	legt fest, daß der Zeichenstift nach Überschreitung der Fenstergrenzen automatisch auf der gegenüberliegenden Seite wieder erscheint

XCor	liefert die momentane X-Koordinate des Zeichenstiftes zurück
YCor	liefert die momentane Y-Koordinate des Zeichenstiftes zurück

13. Der Koprozessor 8087

Turbo Pascal unterscheidet wie die meisten anderen Programmiersprachen zwischen zwei Arten numerischer Werte: Integer (Typen *ShortInt*, *Integer*, *LongInt*, *Byte*, *Word*) und Realzahlen (Typen *Real*, *Single*, *Double*, *Extended*, *Comp*), die aufgrund des internen Speicherformats auch als *Gleitkommazahlen* bezeichnet werden.

Der Prozessor 8086 verfügt über eine ganze Reihe von Befehlen für arithmetische Operationen mit Integern, die Verarbeitung von Realzahlen muß dagegen über (aufwendige und zeitraubende) Unterprogramme erfolgen. Aus diesem Grund wurde der 8087 entworfen – ein spezieller Chip, der Berechnungen mit Gleitkommazahlen über direkte Befehle (und deshalb mit hoher Geschwindigkeit) ausführen kann. Wenn Sie viel mit derartigen Werten arbeiten, sollten Sie sich die Anschaffung dieses Chips überlegen – die Ausführungsgeschwindigkeit eines Programms läßt sich durch einen 8087 erheblich steigern.

Turbo Pascal unterstützt sowohl Systeme mit als auch ohne installiertem Coprozessor:

■ Für Programme, die hauptsächlich auf PCs ohne Coprozessor laufen sollen, bietet die Laufzeitbibliothek den Datentyp *Real* und eine dazugehörige Sammlung von Unterprogrammen, in denen Gleitkomma-Operationen durch den 8086 ausgeführt werden. Eine Variable vom Typ *Real* belegt sechs Bytes Speicherplatz, hat einen zulässigen Wertebereich von $2.9 \cdot 10^{-39}$ bis $1.7 \cdot 10^{38}$ und eine Genauigkeit von 11-12 Dezimalstellen. Die Bibliotheksroutinen sind vor allem hinsichtlich der Geschwindigkeit und des Code-Umfangs optimiert – deshalb fehlen einige spezielle Funktionen, deren Programmierung ohne Coprozessor reichlich aufwendig wäre.

■ Für Programme, bei denen es auf hohe Rechengenauigkeit ankommt (und die vorzugsweise auf Systemen mit einem Coprozessor laufen sollen), kann der Compiler Code erzeugen, der den 8087 – je nach System – entweder direkt ansteuert oder simuliert. Unabhängig von der Ausrüstung des Computers stehen Ihnen damit vier weitere Datentypen (*Single*, *Double*, *Extended* und *Comp*) zur Verfügung, der zulässige Wertebereich geht von $1.9 \cdot 10^{-4951}$ bis $1.1 \cdot 10^{4932}$ mit einer Genauigkeit von 19-20 Dezimalstellen.

Über den Compiler-Befehl *\$N* bzw. den Wahlpunkt *O/C/Numeric processing* kann zwischen den beiden Modi der Code-Erzeugung umgeschaltet werden. *\$N*-steht für die softwaremäßige Behandlung von Gleitkomma-Operationen und ist

die Standardvorgabe, $\$N+$ erzeugt Code zur Ansteuerung des Coprozessors und macht einen weiteren Schalter verfügbar: *O/C/Emulation* bzw. der Compiler-Befehl $\$E$ legt fest, welche Bibliothek für 8087-Operationen in das Programm aufgenommen wird. $\$E+$ steht für die Einbindung eines *Emulators*, der 8087-Befehle nötigenfalls durch 8086-Unterprogramme ersetzt, und ist die Standardvorgabe, $\$E-$ nimmt nur einige Routinen zur direkten Ansteuerung des 8087 auf und macht das Programm um einige Kilobyte kürzer – allerdings läßt es sich dann nur noch auf Systemen verwenden, die tatsächlich mit einem Coprozessor ausgerüstet sind.

Hinweis: Die mathematischen Routinen in *System* (*Pi*, *Sqrt*, *Sin* usw.) liefern Werte vom Typ *Real* zurück, wenn ohne Coprozessor gearbeitet wird ($\{ \$N- \}$). Im Modus $\$N+$ haben die Ergebnisse dagegen den Typ *Extended*:

```
{ $N+ }
begin
  Writeln(Pi); { Ergebnis 3.14159265358979E+0000 }
end.

{ $N- }
begin
  Writeln(Pi); { Ergebnis 3.1415926536E+00 }
end.
```

Hinweis: Der Compiler-Befehl $\$E$ bewirkt nichts, wenn er in einer Unit benutzt wird; er macht nur in einem Programm Sinn. Wenn außerdem das Programm und alle die von ihm benötigten Units im $\{ \$N- \}$ -Modus compiliert wurde, dann bindet Turbo Pascal keine 8087-Emulation ein, und der Compiler-Befehl $\$E$ wird ignoriert.

Die folgenden Abschnitte dieses Kapitels erläutern die Zusammenarbeit zwischen Turbo Pascal und einem numerischen Coprozessor im Detail.

Die Datentypen des 8087

Im Modus $\{ \$N+ \}$ stellt Turbo Pascal vier weitere Datentypen für numerische Werte zur Verfügung:

- Der Typ *Single* ist das »kleinste« Format, in dem sich eine Realzahl speichern läßt: jede Variable dieses Typs belegt vier Bytes Speicherplatz, hat einen Wertebereich von $1.5 \cdot 10^{-45}$ bis $3.4 \cdot 10^{38}$ und eine Genauigkeit von 7 bis 8 Dezimalstellen.
- Der Typ *Double* belegt genau doppelt soviel Speicherplatz wie der Typ *Single*, nämlich 8 Bytes pro Variable. Sein Wertebereich geht von $5.0 \cdot 10^{-324}$ bis $1.7 \cdot 10^{308}$, die Genauigkeit liegt zwischen 15 und 16 Dezimalstellen.
- Der Typ *Extended* stellt das »größte« Format dar, das vom 8087 direkt unterstützt wird: jede Variable dieses Typs belegt 10 Bytes Speicherplatz. Ent-

sprechend gewaltig ist der zulässige Wertebereich, der sich von $1.9 \cdot 10^{-4951}$ bis $1.1 \cdot 10^{4932}$ erstreckt.

■ Der Typ *Comp* speichert ganzzahlige Werte im Bereich von -2^{63} bis $2^{63}-1$ (ca. $-9.2 \cdot 10^{18}$ bis $9.2 \cdot 10^{18}$). Eine Variable dieses Typs belegt 8 Bytes Speicherplatz und ist eigentlich eine Art doppeltegenauer *LongInt* – da *Comp*-Werte aber durch den Coprozessor bearbeitet werden, betrachtet man sie im allgemeinen als Realzahlen. Der Typ *Comp* ist für Berechnungen aus dem Finanzbereich hervorragend geeignet, weil Operationen mit ihm keine Rundungsfehler erzeugen, wenn beispielsweise mit der Grundeinheit »Pfennig« gearbeitet wird.

Der Typ *Real* ist immer verfügbar – unabhängig davon, ob Sie im Modus $\{ \$N+ \}$ oder $\{ \$N- \}$ arbeiten bzw. ob Ihr Computer mit einem Coprozessor ausgerüstet ist oder nicht: Programme, die ohne Unterstützung durch einen 8087 geschrieben wurden, müssen auch für den Betrieb mit einem 8087 nicht modifiziert werden, mit *Real* erzeugte Dateien sind zwischen beiden Code-Modellen uneingeschränkt austauschbar.

Ein Problem soll hier aber nicht verschwiegen werden: Im Modus $\{ \$N+ \}$ werden Operationen mit *Real*-Variablen langsamer ausgeführt – selbst dann, wenn das System mit einem 8087 ausgerüstet ist. Der Grund: Werte dieses Formats können vom Coprozessor nicht direkt verarbeitet werden, vor und nach jeder Berechnung ist daher eine (durch den 8086 ausgeführte und sehr aufwendige) Konvertierung notwendig. Da der Emulator den 8087 zu hundert Prozent simuliert, ist diese Konvertierung im Modus $\{ \$N+ \}$ auch auf Systemen ohne Coprozessor notwendig – hier sinkt die Ausführungsgeschwindigkeit erheblich.

Aus diesem Grund sollten Programme, die nicht ausschließlich für Systeme ohne 8087 geschrieben werden, mit den Datentypen *Single*, *Double*, *Extended* und *Comp* arbeiten.

Arithmetische Operationen im Format Extended

Der Typ *Extended* ist die Grundlage aller Berechnungen, die über den 8087 ausgeführt werden: er wird folglich von Turbo Pascal für die Speicherung von Konstanten und für sämtliche numerischen Ergebnisse verwendet, die keine ganzen Zahlen sind. Im folgenden Beispiel wird also die gesamte rechte Seite der Zuweisung – unabhängig von der Variablen auf der linken Seite – im Format *Extended* berechnet:

```
var
  X, A, B, C: Real;
begin
  X := (B + Sqrt(B * B - A * C)) / A;
end.
```

Diese Automatik hat gleich mehrere Vorteile: Rundungsfehler während der Berechnung können so gut wie vernachlässigt werden, Über- oder Unterschrei-

tungen des Rechenbereichs sind so gut wie ausgeschlossen. Das Ergebnis sind stabilere und zuverlässigere Programme.

Die Eigenschaften des Typs *Extended* lassen sich auch für aufeinanderfolgende Berechnungen ausnutzen, bei denen Turbo Pascal nicht mehr automatisch für eine erhöhte Genauigkeit sorgen kann. Das folgende Beispiel berechnet die Normalbruchreihe, wobei das Ergebnis schließlich als einfachgenaue Realzahl gespeichert wird:

```
var
  Bruchsumme: Single;
  Nenner: Integer;
  T: Extended; { zur Speicherung der Zwischenergebnisse }

begin
  T := 0.0;
  for Nenner := 1 to 20000 do
    T := T + 1.0 / Nenner;
  Bruchsumme := T;
end.
```

(Sie sollten dieses Programm einmal ausführen lassen und mit einer zweiten Version vergleichen, die *T* als *Single* deklariert. In diesem Fall summieren sich die Rundungsfehler bis auf die vierte Stelle hinter dem Komma auf).

Formale Parameter und Funktionsergebnisse können ebenfalls als *Extended* deklariert werden, um unnötige Konvertierungen und die dabei entstehenden Rundungsfehler zu vermeiden. Ein Beispiel:

```
function Flaeche(Radius: Extended): Extended;
begin
  Flaeche := Pi * Radius * Radius;
end;
```

Vergleiche von Realzahlen

Da Realzahlen grundsätzlich *Näherungswerte* darstellen, ergeben Vergleiche zwischen Variablen verschiedener Formate nicht immer die erwarteten Ergebnisse. Ein Beispiel:

```
var
  X: Single;
  Y: Double;

begin
  X := 1/3; Y := 1/3;
  WriteLn(X = Y);
end.
```

Das Ergebnis dieses Programms ist *False*, weil der in *X* gespeicherte Wert auf

7 bis 8 Dezimalstellen gerundet ist, der in *Y* gespeicherte Wert dagegen auf 15 bis 16 Stellen – nach der Konvertierung in das Format *Extended* unterscheiden sich beide etwa ab der 8. Stelle nach dem Komma. Aus demselben Grund ergeben die Befehle

```
X := 1/3;
writeln(X = 1/3);
```

das Ergebnis *False*, wenn *X* als *Single* oder *Double* deklariert ist – Turbo Pascal berechnet den Ausdruck $1/3$ mit einer Genauigkeit von 20 Dezimalstellen.

Der Rechenstack des 8087

Der 8087 verfügt über einen internen Stack, in dem bis zu 8 Werte gespeichert werden können. Zugriffe auf diesen Stack geschehen wesentlich schneller als das Auslesen entsprechender RAM-Zellen – folglich wird er von Turbo Pascal zur Speicherung von Operanden und Zwischenergebnissen verwendet.

Theoretisch ließe sich der Stack des 8087 mit sehr komplizierten Berechnungen zum Überlaufen bringen. Praktisch bildet die Begrenzung auf 8 Werte allerdings kein ernsthaftes Hindernis: Ausdrücke, die mehr als 8 Zwischenergebnisse erzeugen, dürften ausgesprochen selten vorkommen.

Wesentlich reeller sind die Gefahren, die in verschachtelten und/oder rekursiven Funktionsaufrufen liegen – falls derartige Konstrukte nicht solide durchdacht sind, kann der Stack des 8087 sehr wohl überladen werden. Ein Beispiel:

```
X := Test(A,B, Test(C,D, Test(E,F, Text(X,Y,Z))));
```

Hier enthält der Stack beim Aufruf von *Test(X,Y,Z)* bereits sechs Werte – nachdem noch drei weitere dazukommen, ist er überladen. Korrekt programmiert lautet die Zuweisung:

```
X := Test(X,Y,Z);
X := Test(E,F,X);
X := Test(C,D,X);
X := Test(A,B,X);
```

Ein weiteres Beispiel, das aus einer rekursiven Funktion zur Berechnung von Fibonacci-Zahlen besteht:

```
function Fib(N: Integer): Extended;
begin
  if N = 0 then Fib := 0.0
  else if N = 1 then Fib := 1.0
  else Fib := Fib(N-1) + Fib(N-2);
end;
```

Jeder Aufruf dieser Funktion mit einem Argument größer 8 bringt den Stack des 8087 zum Überlaufen. Der Grund: die Berechnung der letzten Zuweisung erfordert die Speicherung eines Zwischenergebnisses (*Fib(N-1)*) – jede Rekursion

belegt also einen Speicherplatz auf dem Stack, der beim neunten Aufruf überläuft. Durch eine korrekte Programmierung wird dieses Problem vermieden:

```
function Fib(N: Integer): Extended;  
var  
    F1, F2: Extended;  
  
begin  
    if N = 0 then Fib := 0.0  
    else if N = 1 then Fib := 1.0  
    else begin  
        F1 := Fib(N-1); F2 := Fib(N-2);  
        Fib := F1 + F2;  
    end;  
end;
```

Hier werden die Zwischenergebnisse als lokale Variablen auf dem Stack des 8086 gespeichert (der natürlich auch irgendwann einmal überläuft – dazu sind aber recht große Verschachtelungstiefen nötig).

Die Ausgabe von Realzahlen

Weil Turbo Pascal bei der Verwendung eines 8087 grundsätzlich im Format *Extended* arbeitet, geben *Write* und *Writeln* nicht zwei, sondern vier Ziffern für den Exponenten aus, um dem erweiterten Rechenbereich gerecht zu werden. Aus demselben Grund erzeugt die Standard-Prozedur *Str* Strings mit vier Stellen für den Exponenten.

Units und der 8087

Units, die den 8087 bzw. den Emulator verwenden, können nur dann in Programme aufgenommen werden, wenn auch die Compilierung des Hauptmoduls im Modus *\$N+* stattfindet. Turbo Pascal prüft allerdings bei der Aufnahme einer Unit unabhängig von darin enthaltenen *\$N*-Befehlen, ob tatsächlich 8087-Code enthalten ist – falls nein, spielt der Stand des Schalters *O/C/Numeric processing* keine weitere Rolle.

Maschinenprogramme und der Emulator

Die Simulation des Coprozessors durch den Emulator ist nicht auf Pascal-Routinen beschränkt – sie funktioniert auch für (via $\{ \$L \langle Dateiname \rangle \}$ eingebundene) Maschinenprogramme mit direkten 8087-Befehlen. Die einzige Voraussetzung ist ein zusätzlicher Kommandozeilen-Parameter für den Assembler bei der Übersetzung: bei einem Aufruf von MASM bzw. TASM wie

```
MASM /E TEST;      <- Übersetzung von TEST.ASM in TEST.OBJ
TASM /E TEST;      <- dito
```

prüfen beide Assembler, ob der Quelltext 8087-Befehle enthält – wenn ja, werden diese Befehle speziell gekennzeichnet und damit für den Linker von Turbo Pascal verfügbar gemacht.

Achtung: Bei der Übersetzung einer .ASM-Datei ohne den Parameter /E unterbleibt diese Kennzeichnung – derartige Routinen setzen (wie in den schlechten alten Zeiten ohne Emulator) einen Coprozessor voraus und produzieren Unsinn bzw. Systemabstürze, wenn kein 8087 installiert ist.

Die Prüfung des 8087 beim Start des Programms

findet normalerweise automatisch statt, wenn das Programm im Modus $\{ \$N+ \}$ compiliert wurde. In diesem Fall enthält der Initialisierungsteil von *System* eine zusätzliche Routine, die feststellt, ob ein Coprozessor installiert ist oder nicht:

- falls ein 8087 vorhanden ist, wird er (mit direkten Befehlen) benutzt;
- falls kein 8087 vorhanden ist, hängt der weitere Programmverlauf vom Stand des Schalters $\$E$ bei der Compilierung ab: Im Modus $\{ \$E+ \}$ übernehmen die Unterprogramme des Emulators sämtliche Operationen mit Gleitkommazahlen; im Modus $\{ \$E- \}$ – der einen 8087 voraussetzt – bricht die Prüfroutine das Programm mit einer Fehlermeldung ab.

Was wir noch nicht definiert haben, ist der Begriff »vorhanden«:

- die Modelle PC und XT verfügen über einen DIP-Schalter (Mäuseklavier) auf der Hauptplatine. Ob ein Coprozessor installiert ist oder nicht, wird über eine Abfrage der Schalterstellung geprüft;
- der PCjr ist in dieser Hinsicht nicht erweiterbar, aber »kompatibel« – eine »Abfrage« des nicht vorhandenen Schalters ergibt immer »kein Coprozessor«;
- Computer der AT-Serie und »Kompatible« werden durch eine Reihe direkter Tests geprüft, weil hier zwischen den Möglichkeiten »80287« und »80387« unterschieden werden muß.

Falls es (speziell auf PC-Nachbauten) irgendwelche Probleme bei der Prüfung

des 8087 geben sollte, bietet Turbo Pascal eine zusätzliche Möglichkeit: Bevor die Routinen von *System* tatsächlich über eine Abfrage des Systems das Vorhandensein eines Coprozessors prüfen, wird die MSDOS-Tabelle *Environment* ausgelesen. Findet sich in dieser Tabelle ein Eintrag der Form

87=Y oder 87=N

dann wird dieser Eintrag bewertet, eine weitere Abfrage des Systems findet nicht statt. Die Tabelle *Environment* wird über den DOS-Befehl SET abgefragt, verändert, gelöscht oder erweitert:

SET gibt sämtliche Einträge der Tabelle aus;

SET xxx=yyy sucht die Tabelle nach dem Eintrag *xxx* ab. Wenn dieser Eintrag bereits existiert, wird er auf den Wert *yyy* verändert, ansonsten wird ein neuer Eintrag mit dem Wert *yyy* erzeugt;

SET xxx= löscht den Eintrag *xxx* in der Tabelle (soweit vorhanden).

Ein einmal gesetzter Eintrag in *Environment* bleibt solange erhalten, bis Sie ihn entweder mit einem weiteren SET-Befehl löschen oder den Computer neu starten.

Wenn Sie also festlegen wollen, daß kein Coprozessor installiert ist, dann geben Sie vor dem ersten Start eines Turbo Pascal-Programms den DOS-Befehl

SET 87=N

Besser wäre natürlich noch, diesen Befehl gleich in die Datei AUTOEXEC.BAT aufzunehmen – er wird dann bei jedem Start des Computers automatisch ausgeführt.

In umgekehrter Richtung funktioniert das Verfahren über SET ebenfalls – auch wenn es noch etwas unwahrscheinlicher ist, daß Sie einen Coprozessor installiert haben, den das System nicht als solchen erkennen mag. Nach dem DOS-Befehl

SET 87=Y

führt jedes Turbo Pascal-Programm ebenfalls keine weiteren Tests mehr aus und verläßt sich darauf, daß ein Coprozessor installiert ist.

Vorsicht: Niemand hält Sie davon ab, einen solchen Befehl auch auf einem System ohne Coprozessor zu geben. SET ist eine reine Verwaltung von *Environment* und führt mitnichten irgendwelche Prüfungen aus – Sie können beispielsweise auch den Parameter *HOPP=TOP!* eintragen, ohne irgendeine Reaktion zu erhalten.

Ein Turbo Pascal-Programm, das sich auf einen nicht vorhandenen Coprozessor verläßt, bringt den Computer völlig durcheinander.

... und ihr Ergebnis: Die Systemvariable Test8087

Die Unit *System* definiert eine Variable namens *Test8087*, die durch die

Prüfung auf »Coprozessor vorhanden« einen der folgenden vier Werte zugewiesen bekommt:

- 0 kein Coprozessor erkannt
- 1 8087
- 2 80287
- 3 80387

Diese Unterscheidung ist notwendig, weil zwischen den drei Versionen des Coprozessors einige kleinere Inkompatibilitäten bestehen, auf die hauptsächlich die Bibliotheksroutinen zur Behandlung von Gleitkomma-Fehlern Rücksicht nehmen müssen. *Test8087* ist als globale Variable vom Typ *Byte* deklariert, wird auch im Modus *{ E -}* gesetzt und läßt sich jederzeit von einem Pascal-Programm aus abfragen. (Damit läßt sich also *in* einem Programm feststellen, ob momentan über die Routinen des Emulators oder mit direkten 8087-Befehlen gearbeitet wird).

8087-Emulation mit dem Assembler

Wenn eine Object-Datei mit *{ L Dateiname}* eingebunden wird, muß sichergestellt sein, daß das Object mit der eingeschalteten 8087-Emulation übersetzt wurde. Wenn Sie etwa 8087-Befehle in einer Assembler **external**-Prozedur benutzen, müssen Sie beim Assemblieren die 8087-Emulation einschalten, sonst können die 8087-Befehle auf Rechnern ohne 8087-Prozessor nicht emuliert werden. Die 8087-Emulation wird bei Turbo Assembler mit der Kommandozeilen-Option */e* eingeschaltet.

14. Interne Details

Dieses Kapitel ist für erfahrene Spezialisten gedacht und enthält eine Beschreibung der technischen Hintergründe: Aufteilung des Speichers und seine dynamische Verwaltung, verwendete Datenformate, Abläufe bei Aufrufen von Prozeduren und Funktionen und schließlich mögliche Optimierung des Programmcodes.

Speicherbelegung durch ein Turbo-Pascal-Programm

Turbo-Pascal-Programme werden als .EXE-Dateien gespeichert und bekommen beim Laden von der Diskette durch DOS ein Programmsegment-Präfix (PSP) vorangestellt. Das PSP umfaßt 256 Bytes, seine Segment-Adresse ist in der vordefinierten Variablen *PrefixSeg* (Typ: *Word*) gespeichert.

Das Programm und jede aufgenommene Unit verfügen über ein eigenes Code-Segment, wobei das Programm selbst auf der niedrigsten Adresse steht, darauf folgen die Units in der *umgekehrten* Reihenfolge ihrer Angabe durch **uses**. Das Code-Segment mit der höchsten Adresse wird grundsätzlich von der Laufzeitbibliothek (d.h. der Unit *System*) belegt. Die Größe eines einzelnen Segments (und damit die Größe einer einzelnen Unit und des eigentlichen Programms) ist auf 64 KByte beschränkt. Da ein Programm beliebig viele Units aufnehmen kann, wird seine *Gesamtgröße* aber nur durch den verfügbaren Hauptspeicher begrenzt.

Das Daten-Segment wird über das Prozessor-Register DS adressiert und nimmt alle typisierten Konstanten sowie alle globalen Variablen auf. Der Wert des DS-Registers bleibt während der Ausführung eines Programms konstant: globale Variablen und typisierte Konstanten dürfen also zusammen maximal 64 KByte Speicherplatz belegen.

Beim Start des Programms wird das Registerpaar SS:SP (Stack-Segment und Stackzeiger) so gesetzt, daß es auf das erste Byte oberhalb des für den Stack reservierten Bereichs zeigt. Das Register SS bleibt während der gesamten Ausführung eines Programms konstant, maximal stehen deshalb 64 KByte für den Stack zur Verfügung. Die Speicherung von Werten auf dem Stack *erniedrigt* das SP-Register, belegt also absteigende Speicheradressen: ein Stack-Überlauf besteht also darin, daß SP die Anfangsadresse des Stack-Bereichs (SS:\$0000) *unter-*

schreitet. Turbo Pascal reserviert standardmäßig 16 KByte Speicherplatz für den Stack; dieser Wert kann mit dem Compiler-Befehl `$M` geändert werden.

Abbildung 14.1 gibt die Aufteilung des Speichers durch ein Turbo-Pascal-Programm wieder:

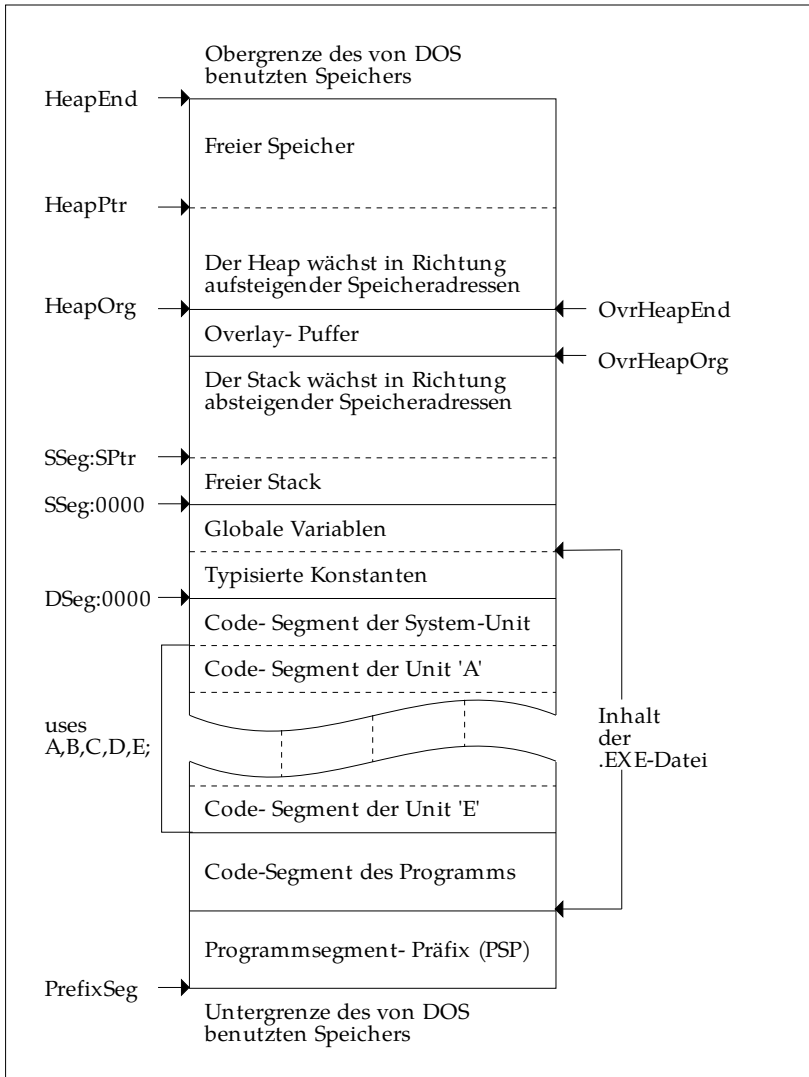


Abbildung 14.1 Das Speicherlayout eines Turbo-Pascal-Programms

Der Overlay-Puffer existiert nur dann in physikalischer Form, wenn das Programm mit Overlays arbeitet (und die Unit *Overlay* mit **uses** aufgeführt hat). In diesem Fall wird für diesen Puffer standardmäßig soviel Speicherplatz reser-

viert, daß die größte als Overlay deklarierte Unit genau hineinpaßt; über die Prozedur *OvrSetBuf* läßt sich der Puffer vergrößern (wobei sich der Beginn des regulären Heaps (auf den *HeapOrg* zeigt) entsprechend in Richtung höherer Speicheradressen verschiebt). Für Programme, die ohne *Overlay* arbeiten, hat der Overlay-Puffer die Größe 0.

Über die Standard-Prozeduren *New* und *GetMem* erzeugte Variablen werden als *dynamisch* bezeichnet und im *Heap-Bereich* des Computers gespeichert. Als Heap wird der gesamte restliche Speicher des Computers bezeichnet, der nicht durch das Betriebssystem und das laufende Programm belegt ist. Über den Compiler-Befehl *\$M* läßt sich (zusammen mit der Stackgröße) eine Minimal- und eine Maximalgröße für den Heap festlegen. Die Standardvorgabe für die Minimalgröße ist 0 – wird sie erhöht, dann nimmt Turbo Pascal beim Start des Programms eine entsprechende Prüfung vor und bricht ab, wenn die angegebene Menge an Speicherplatz nicht zur Verfügung steht. Die Standardvorgabe für die Maximalgröße ist 1 Megabyte, d.h. der gesamte freie Speicher wird als verfügbar betrachtet.

Zu diesem Thema gibt es noch so viel zu sagen, daß wir ihm einen eigenen Abschnitt gewidmet haben.

Die dynamische Verwaltung des Speichers

Der Heap-Bereich wird von Turbo Pascal als eine Art Stack behandelt – er wächst allerdings in Richtung *aufsteigender* Speicheradressen. Seine Startadresse ist in der globalen Variablen *HeapOrg* (Heap Origin) gespeichert und seine Endadresse in *HeapEnd*. Seine momentane *Spitze* – also die Startadresse des noch freien Speichers – findet sich in der Variablen *HeapPtr*. Die »dynamische Belegung von Speicherplatz« durch *New* oder *GetMem* besteht aus drei einfachen Schritten:

- Prüfung, ob oberhalb der momentanen Spitze des Heaps noch genügend Speicherplatz vorhanden ist;
- Erhöhung von *HeapPtr* um die Größe der neuen dynamischen Variablen;
- Zurückliefern des *alten* Wertes von *HeapPtr* als Zeiger auf die neue dynamische Variable. Dieser Zeigerwert enthält damit die Startadresse des neu belegten Bereichs.

Durch dieses Arbeitsprinzip werden nacheinander erzeugte dynamische Variablen also quasi aufeinander gestapelt.

HeapPtr wird nach jeder Veränderung automatisch *normalisiert*, d.h. so umgerechnet, daß der Offset-Anteil im Bereich von \$0000 bis \$000F liegt. Die maximale Größe einer einzelnen dynamischen Variablen ist auf 65521 Bytes (\$10000 minus \$000F) begrenzt, weil jede Variable vollständig innerhalb eines einzigen Segments gespeichert sein muß.

Freigabe von dynamischem Speicherplatz

Turbo Pascal bietet zwei grundsätzlich verschiedene Verfahren, um zuvor belegte Teile des Heaps wieder freizugeben: *Dispose* und *FreeMem* oder *Mark* / *Release*. Die zuletzt genannte Methode ist die einfachere, wie das folgende Beispiel zeigt:

```
New(Zeiger1);
New(Zeiger2);
Mark(P);
New(Zeiger3);
New(Zeiger4);
New(Zeiger5);
Readln;
Release(P);
```

Abbildung 14.2 gibt den Zustand des Heaps vor der Ausführung von *Readln* wieder – also nach der Erzeugung von fünf dynamischen Variablen:

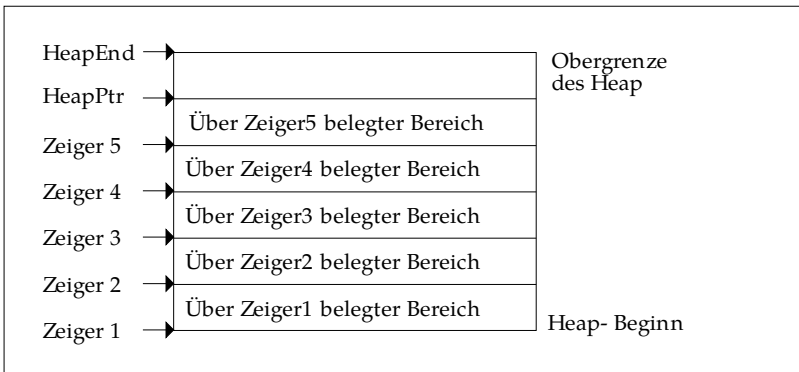


Abbildung 14.2 Fünf dynamische Variablen

Der Befehl *Mark(P)* hat auf den Inhalt des Heaps keine Wirkung – er hält in unserem Beispiel den Stand von *HeapPtr* direkt vor der nächsten Belegung (mit *New(Zeiger3)*) fest. Mit *Release(P)* wird der Bereich oberhalb von *P* wieder freigegeben (siehe Abbildung 14.3).

Release(P) weist *HeapPtr* den Wert der Zeigervariablen *P* zu und setzt so die Spitze des Heaps wieder herab. Das effektive Ergebnis dieses Befehls ist in unserem Beispiel also die Freigabe der über *Zeiger3* bis *Zeiger5* belegten Speicherbereiche. In derselben Weise gibt der Befehl

```
Release(HeapOrg);
```

den gesamten Heap wieder frei (löscht also alle dynamischen Variablen), weil *HeapOrg* auf die unterste für den Heap verfügbare Adresse zeigt.

Vorsicht: Das Grafikpaket belegt einige Kilobytes für interne Puffer, Grafik-

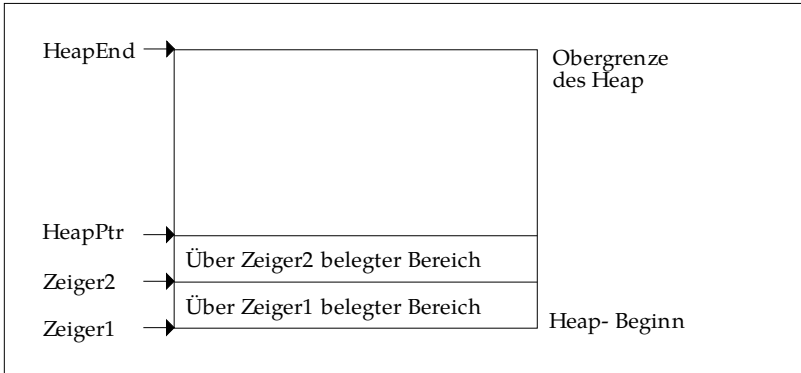


Abbildung 14.3 Der Heap nach dem Befehl *Release(P)*

treiber und Zeichensätze auf dem Heap. Die entsprechenden dynamischen Variablen werden über einen Aufruf von *GetMem* erzeugt – also ohne jedes Hintertürchen. Ergo: An einem Programm, das den Heap mit *Release(HeapOrg)* nach der Initialisierung via *InitGraph* komplett »abräumt«, wird man nicht viel Freude haben.

Achtung: *Release* verändert die Werte der Zeigervariablen (in unserem Beispiel *Zeiger3* .. *Zeiger5*) nicht – sie zeigen nach wie vor auf die nunmehr »freigegebenen« Speicherbereiche. Die *Inhalte* dieser Bereiche werden durch weitere Definitionen dynamischer Variablen verändert: Leseoperationen über zuvor freigegebene Zeigervariablen liefern danach meistens undefiniertes, Schreiboperationen bringen unter Umständen die Speicherverwaltung (und damit das System) zum Absturz.

Für Anwendungen, die Zeigervariablen genau in der umgekehrten Reihenfolge ihrer Erzeugung wieder freigeben, sind *Mark* und *Release* eine ausgesprochen effiziente Lösung – in den meisten praktischen Fällen werden Belegung und Freigabe des Heaps aber etwas ungeordnet geschehen und deshalb eine komplexere Art der Verwaltung mit *Dispose* und *FreeMem* notwendig machen. Diese beiden Prozeduren erlauben die Freigabe isolierter Speicherbereiche (also einzelner dynamischer Variablen) zu beliebigen Zeitpunkten.

Wenn eine Variable über *Dispose* oder *FreeMem* freigegeben wird, die nicht an der momentanen Spitze des Heaps steht, ergibt sich ein entsprechendes »Loch«. Zur Demonstration verwenden wir – leicht modifiziert – noch einmal das zuvor gezeigte Beispiel:

```
New(Zeiger1);
New(Zeiger2);
Mark(P);
New(Zeiger3);
New(Zeiger4);
New(Zeiger5);
Readln;
Dispose(Zeiger3)); { gibt eine Variable in der "Mitte" wieder frei }
```

Nach der Freigabe des über *Zeiger3* belegten Bereichs sieht der Heap so aus wie in Abbildung 14.4 gezeigt:

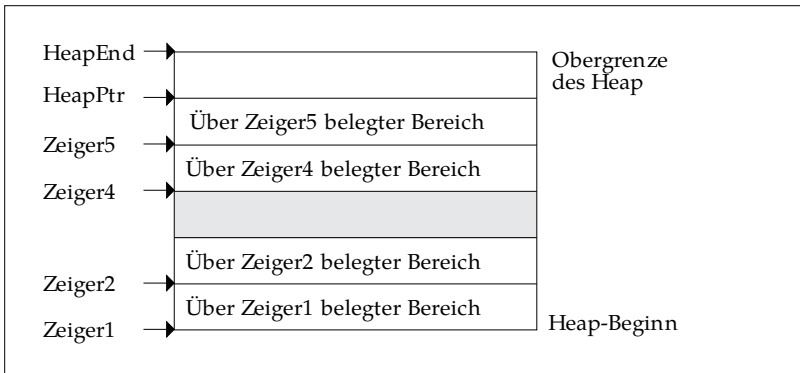


Abbildung 14.4 Erzeugung eines »Lochs« im Heap

Wenn wir zu diesem Zeitpunkt den Befehl *New(Zeiger3)* ein weiteres Mal geben, dann wird das Loch wieder belegt – die Spitze des Heaps bleibt unverändert. Dasselbe ist bei der Erzeugung einer dynamischen Variablen der Fall, solange sie in dieses Loch hineinpaßt. Anders gesagt: solange der Heap *fragmentiert* ist (also Löcher enthält), versuchen *New* und *GetMem* immer zuerst, diese Löcher aufzufüllen, bevor *HeapPtr* weiter erhöht wird.

Ein weiterer Mechanismus: wenn wir als nächstes den über *Zeiger4* belegten Speicherbereich freigeben, entstehen zwei nebeneinanderliegende Löcher, die automatisch zu einem Bereich zusammengefaßt werden:

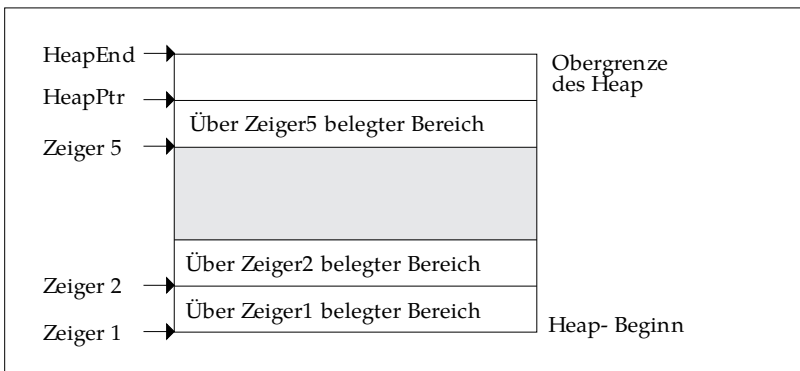


Abbildung 14.5 Der Heap nach dem Befehl *Dispose(Zeiger5)*

Nachdem die über *Zeiger3* und *Zeiger4* belegten Bereiche bereits freigegeben sind, führt der Befehl

```
Dispose(Zeiger5);
```

automatisch zwei aufeinanderfolgende Schritte aus: erstens die Freigabe des entsprechenden Bereichs und das Verbinden der L cher; zweitens wird *HeapPtr* entsprechend herabgesetzt – n mlich auf die erste oberhalb des Bereichs von *Zeiger2* verf gbare Adresse.

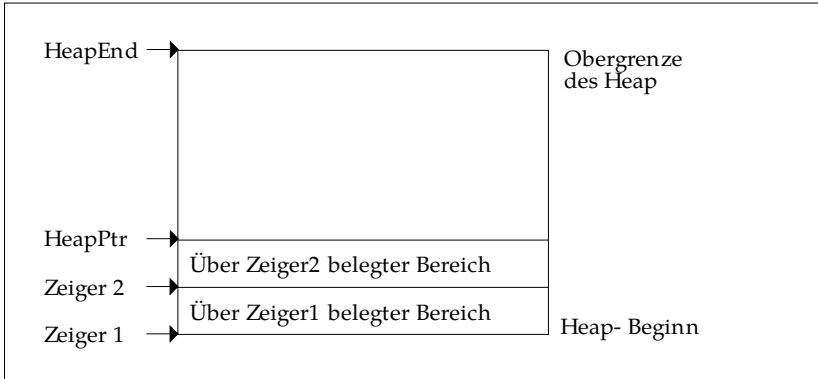


Abbildung 14.6 Herabsetzen von *HeapPtr* als Folge von *Dispose*

Der Effekt ist letztendlich derselbe wie bei einer Freigabe  ber *Release(P)* (vgl. Abbildung 14.2), allerdings wurden die durch das Programm erzeugten L cher in einer *Fragmentliste* f r eine eventuelle neue Verwendung festgehalten.

Die Fragmentliste

Der Heapmanager von Turbo Pascal 6.0 verwendet eine andere Technik als der der Version 5.5, um sich unbelegte Speicherbereiche zu merken. In der Version 5.5 gab es eine *Fragmentliste*, die Verweise auf unbenutzte Speicherbl cke enthielt und von der Obergrenze des zur Verf gung stehenden Speichers nach unten (gegen *HeapPtr*) wuchs. Dabei wurde Speicher in 1-Byte-Schritten vergeben.

Dies ist in der Version 6.0 nicht mehr der Fall. Die Zuteilung von Speicher erfolgt nun in Schritten von 8 Bytes. Diese  nderung hat gute Gr nde. Bei Programmen, die extensiv vom Heap Gebrauch machen und dabei st ndig unterschiedlich gro e Speicherbl cke anfordern, wird der Heap sehr stark fragmentiert. Insbesondere objektorientierte Programme, die sehr viel mit dynamischen Strukturen arbeiten, sind davon betroffen. Haben n mlich dynamische Objekte aus einer Hierarchie fast dieselbe Gr  e, ist die »Trefferquote« bei einer Granularit t von 8 Bytes wesentlich gr  er. Der gr  te Nachteil einer starken Fragmentierung liegt allerdings nicht in der schlechten Speicherausnutzung, sondern in der Tatsache, da  die *Fragmentliste* soweit anwachsen kann, da  die Freigabe eines Blockes nicht mehr m glich ist, weil f r den daf r notwendigen Eintrag kein Platz mehr vorhanden ist. Deshalb kennt Turbo Pascal 6.0 keine *Fragmentliste* mehr, was nat rlich bei Programmen, die den Heap direkt mani-

pulieren, zu Kompatibilitätsproblemen führen kann. Die freien Speicherblöcke werden nun – ähnlich wie in C – selbst in einer Liste verkettet. Die Variable *FreeList* zeigt dabei auf den ersten freien Block. Sind keine Blöcke in dieser Liste gespeichert ist *FreeList* gleich *HeapPtr*.

Um die Verkettung von freien Blöcken und deren Wiederverwendung zu ermöglichen, wird in den ersten 8 Bytes des freien Blocks ein Zeiger auf den nächsten Block sowie eine Größenangabe gespeichert. Dabei gilt folgendes Layout:

```
type
  PFreeRec = ^TFreeRec;
  TFreeRec = record
    Next: PFreeRec;
    Size: Pointer;
  end;
```

Das Feld *Next* zeigt dabei auf den nächsten freien Block oder auf *HeapPtr*, wenn es sich um den letzten freien Block in der Kette handelt. Das Feld *Size* gibt die Größe des freien Blocks an. Allerdings ist diese Größenangabe kein »normaler« 32-Bit-Wert. Vielmehr handelt es sich um eine Art normalisierter Zeiger. Das höherwertige Wort enthält die Anzahl von Paragraphen (16 Bytes) und das niederwertige die Anzahl von Bytes, die dieser Block groß ist. Um die Größe als Longint-Wert zu ermitteln, kann folgendes Berechnungsschema verwendet werden:

```
function BlockSize(Size: Pointer): Longint;
type
  PtrRec = record Lo, Hi: Word
end;
begin
  BlockSize := Longint(PtrRec(Size).Hi)*16+PtrRec(Size).Lo;
end;
```

Um sicherzustellen, daß immer genug Speicher für eine Struktur vom Typ *TFreeRec* am Anfang eines Speicherblocks ist, der freigegeben werden soll, weist Turbo Pascal deshalb den Speicher in Schritten von acht Bytes zu. Mit anderen Worten: der Heap-Manager rundet jede Speicheranforderung auf ein Vielfaches von acht auf. Dies bedeutet, daß acht Bytes vergeben werden, wenn die angeforderte Größe zwischen einem und acht Bytes liegt, 16 Bytes bei einer Größe zwischen neun und 16 und so weiter. Diese Technik scheint auf den ersten Blick etwas verschwenderisch, weil bei einer angeforderten Größe von nur einem Byte jeweils acht Bytes vergeben werden. Allerdings wird Speicher gewöhnlich in größeren Blöcken angefordert. Der Vorteil einer größeren Granularität wird bei einem Editor, der seine Zeilen dynamisch verwaltet, deutlich: Wird zum Beispiel ein 50 Bytes großer Block freigegeben, weil eine Zeile gerade verändert wurde, wird jede Anforderung zwischen 49 und 56 Bytes den gerade freigewordenen Block vollständig verwenden können. Würde immer genau die angeforderte Größe verwendet, blieben sehr viele kleine Speicherbereiche ungenutzt, die den Heap sehr stark fragmentierten.

Die Variable HeapError

zeigt normalerweise auf die in Turbo Pascal eingebaute Fehlerbehandlung – wenn *New* oder *GetMem* ihre Aufgabe nicht ordnungsgemäß erfüllen können, führen sie einen Aufruf über *HeapError* aus. Die Funktion, auf die *HeapError* standardmäßig zeigt, ist folgendermaßen deklariert:

```
function HeapFunc(Size: Word): Integer; far;
```

Hinweis: Die **far**-Direktive legt fest, daß *HeapFunc* als *far* aufgerufen wird.

Die Installation dieser Routine geschieht durch eine explizite Zuweisung an *HeapError*:

```
HeapError := @HeapFunc;
```

HeapFunc bekommt im Fehlerfall die Größe des zu belegenden Bereichs als Parameter *Size* übergeben und kann nur drei verschiedene Ergebnisse haben:

- der Wert 0 erzeugt einen Laufzeitfehler und damit den sofortigen Abbruch des Programms;
- der Wert 1 steht zwar ebenfalls für eine Fehlerbedingung, bricht das Programm aber nicht ab, sondern bewirkt, daß *New* und *GetMem* den Zeigerwert **nil** zurückliefern;
- der Wert 2 steht für »Wiederholung«. *New* und *GetMem* versuchen in diesem Fall erneut, ihre Aufgabe auszuführen (was einen weiteren Aufruf über *HeapError* zur Folge haben kann).

Die in Turbo Pascal eingebaute Variante von *HeapFunc* liefert immer das Ergebnis 0, bewirkt also den Abbruch des Programms durch einen Laufzeitfehler. Sie läßt sich beispielsweise durch die folgende einfache Routine ersetzen, die für die meisten Anwendungen bereits ausreichend ist:

```
function HeapFunc(Size: Word): Integer; far
begin
  HeapFunc := 1;
end;
```

Nach der Installation dieser Version über eine Zuweisung an *HeapError* liefern *New* und *GetMem* im Fehlerfall den Zeigerwert **nil** zurück, anstatt das Programm abzubrechen.

Hinweis: Ein Aufruf der *HeapError*-Funktion mit dem Parameter 0 zeigt an, daß der Heap-Manager gerade *HeapPtr* erhöht hat, um Platz für den eben angeforderten Speicherblock zu schaffen. Diese Situation entsteht immer dann, wenn kein freier Block ausreichender Größe in der Kette der freien Speicherblöcke gefunden wurde. Diese Art Aufruf mit der Größe 0 stellt deshalb keinen Fehler dar, weil durch Verschieben von *HeapPtr* genügend Platz geschaffen wurde, um der Speicheranforderung gerecht zu werden. Aus diesem Grund ignoriert der Heap-Manager in diesem Fall auch das zurückgegebene Funktionsergebnis. Dieses Verhalten kann allerdings – wie zum Beispiel in Turbo Vision

– dazu verwendet werden, um ein konkurrierendes, zweites Heap-Management zu installieren, welches den freien Bereich zwischen *HeapPtr* (der Spitze des Heaps) und *HeapEnd* (dem Ende des verwendbaren Bereichs) benutzt. Immer, wenn der Heap-Manager durch die Größenangabe 0 zeigt, daß sich der Bereich verkleinert hat, kann der alternative Manager darauf reagieren.

Turbo Vision verwendet den Speicherbereich zwischen *HeapPtr* und *HeapEnd* für Puffer zur Darstellung von View-Objekten. Um eine höhere Geschwindigkeit zu erzielen, wird die optische Repräsentation beim Verschieben oder Neuzeichnen deshalb in einem solchen Puffer abgelegt. Braucht das Programm selbst aber mehr Heap-Speicher, wird der Puffer-Manager vom Heap-Manager durch einen entsprechenden Aufruf der *HeapError*-Routine in Kenntnis gesetzt. Danach muß sich der Puffer-Manager mit einem kleineren (bzw. gar keinem) Speicher zufrieden geben.

Interne Datenformate

Integer- Typen

Das von Turbo Pascal verwendete Format zur Darstellung eines Integerwertes hängt von den unteren und oberen Grenzen des erforderlichen Wertebereichs ab:

- Wenn beide Grenzen innerhalb des Bereichs -128..127 liegen (Datentyp *ShortInt*), dann wird ein einzelnes Byte zur Speicherung verwendet, dessen höchstwertiges Bit als Vorzeichen dient.
- Wenn beide Grenzen innerhalb des Bereichs 0..255 liegen (Datentyp *Byte*), dann verwendet Turbo Pascal ein einzelnes Byte zur Speicherung, dessen Wert grundsätzlich als positiv angenommen wird (vorzeichenlos).
- Wenn beide Grenzen innerhalb des Bereichs -32768..32767 liegen (Datentyp *Integer*), dann werden zwei Bytes zur Speicherung verwendet, das höchstwertige Bit des Bytes mit der höheren Adresse dient als Vorzeichen.
- Wenn beide Grenzen innerhalb des Bereichs 0..65535 liegen (Datentyp *Word*), dann verwendet Turbo Pascal zwei Bytes, deren Wert grundsätzlich als positiv angenommen wird.
- In allen anderen Fällen wird ein *LongInt* (vier Bytes, höchstwertiges Bit = Vorzeichen) zur Speicherung verwendet.

Char-Typen

Werte des Typs *Char* oder Teilbereiche davon werden als vorzeichenloses Byte (also wie der Datentyp *Byte*) gespeichert.

Boolesche Typen

Werte des Typs *Boolean* werden mit einem einzelnen Byte gespeichert, das nur die Werte 0 (=False) und 1 (=True) annehmen kann.

Aufzählungstypen

werden als vorzeichenloses Byte (wie der Datentyp *Byte*) gespeichert, wenn die Aufzählung weniger als 256 verschiedene Werte umfaßt. Ansonsten verwendet Turbo Pascal das Speicherformat *Word*.

Real-Typen

speichern die binäre Repräsentation einer Gleitkommazahl in drei voneinander getrennten Teilen: dem Vorzeichen (+ oder -), dem *Exponenten* und der *Mantisse*. Diese Unterteilung gilt für sämtliche Real-Typen, also *Real*, *Single*, *Double*, *Extended* und *Comp*. Allgemein läßt sich dieses Format so darstellen:

+/- Mantisse * 2^{Exponent}

Der Dezimalpunkt wird dabei nicht eigens gespeichert, sondern implizit angenommen: Er steht grundsätzlich rechts des höchstwertigen Bits (MSB = »most significant bit«) – die Mantisse hat also immer einen Wert im Bereich von größer/gleich 0 und kleiner als 2.

Hinweis: In allen Real-Formaten wird das höchstwertige Byte (das den impliziten Dezimalpunkt enthält) auf der höchsten Adresse gespeichert. Darauf folgen die restlichen Bytes der Mantisse in Richtung absteigender Adressen, schließlich kommt der Exponent (in dem zusätzlich das Vorzeichen gespeichert ist).

Der Datentyp Real

belegt 6 Bytes (48 Bit), die entsprechend der Abbildung 14.7 auf der folgenden Seite angeordnet sind:

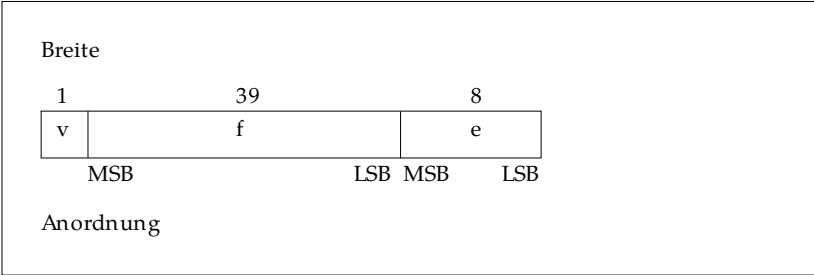


Abbildung 14.7 *Das Format des Datentyps Real*

Der Wert *w* einer in diesem Format gespeicherten Zahl lässt sich folgendermaßen berechnen:

```
if 0 < e <= 255 then w = (-1)v * 2( e-127 ) * (1.f)
if e = 0 then w = 0
```

Hinweis: Im Format *Real* können weder NaNs (Not a Number« = undefinierte Werte) noch denormalisierte Zahlen oder Unendlichkeiten gespeichert werden: denormalisierte Zahlen werden gleich 0 gesetzt, NaNs und Unendlichkeiten erzeugen einen Überlauf.

Der Datentyp Single

belegt 4 Bytes (32 Bit), die folgendermaßen angeordnet sind:

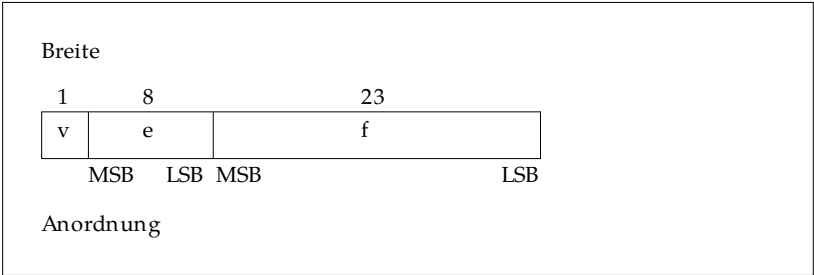


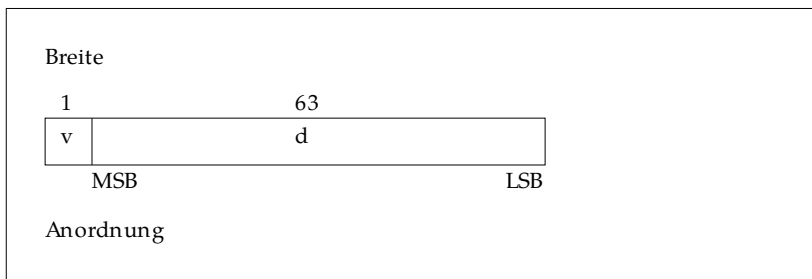
Abbildung 14.8 *Das Format des Datentyps Single*

Der Wert *w* einer in diesem Format gespeicherten Zahl lässt sich folgendermaßen berechnen:

```
if 0 < e and e < 255 then w = (-1)v * 2( e-127 ) * (1.f)
if e = 0 and f <> 0 then w = (-1)v * 2( -126 ) * (0.f)
if e = 0 and f = 0 then w = (-1)v * 0
if e = 255 and f = 0 then w = (-1)v * Inf
if e = 255 and f <> 0 then w = NaN
```

Der Datentyp Double

belegt 8 Bytes (64 Bit), die folgendermaßen angeordnet sind:

Abbildung 14.11 Das Format des Datentyps `Comp`

Der Wert w einer in diesem Format gespeicherten Zahl lässt sich folgendermaßen berechnen:

if $v = 1$ **and** $d = 0$ **then** $w = \text{NaN}$

Ansonsten ist w der durch die 64 Bit im Zweierkomplement angegebene Wert.

Zeiger-Typen

werden mit zwei aufeinanderfolgenden Wörtern (Datentyp `Word`) gespeichert: auf der niedrigeren Adresse steht der Offset-Anteil, auf der höheren Adresse der Segment-Anteil.

Die Zeiger-Konstante **nil** hat den Wert `$0000:$0000`, also gilt `Longint(nil) = 0`.

String-Typen

Ein String wird als Folge einzelner Bytes (Datentyp `Char`) gespeichert, jedes Zeichen belegt also ein Byte. Turbo Pascal stellt die momentane Länge eines Strings in einem weiteren Byte voran, der Gesamtbedarf zur Speicherung beträgt deshalb »maximale Stringlänge plus 1« Bytes. Das Längenbyte steht auf der niedrigsten Speicheradresse, darauf folgen die momentan in dem String gespeicherten Zeichen. Sowohl die Länge als auch die gespeicherten Werte werden als vorzeichenlos betrachtet. Aus der Tatsache, daß die momentane Länge eines Strings in einem Byte gespeichert wird, ergibt sich auch seine Maximallänge: sie beträgt 255 Zeichen.

Mengen

werden als Bit-Array gespeichert, wobei die Position des jeweiligen Bits für die Ordinalität und sein Wert für das Vorhandensein des entsprechenden Elements steht (1« = Element vorhanden, »0« = nicht vorhanden). Eine Menge

kann maximal 256 Elemente enthalten und benötigt maximal 32 Bytes Speicherplatz ($32 * 8 = 256$). Der tatsächliche Speicherplatzverbrauch läßt sich mit der folgenden Formel berechnen:

$$\text{ByteZahl} := (\text{Max} \text{ div } 8) - (\text{Min} \text{ div } 8) + 1;$$

Max steht hier für das Element mit der höchsten, *Min* für das mit der niedrigsten Ordinalzahl. Die Position eines Elements mit der Ordinalzahl *E* innerhalb der Menge ergibt sich aus:

$$\text{ByteNummer} := (E \text{ div } 8) - (\text{Min} \text{ div } 8);$$

$$\text{BitNummer} := E \text{ mod } 8;$$

Array-Typen

werden als kontinuierliche Folge ihrer Elemente gespeichert, wobei das Element mit dem niedrigsten Index auf der niedrigsten Adresse steht. Mehrdimensionale Arrays werden so gespeichert, daß die Elemente der »äußersten rechten« (also letzten) Dimension direkt aufeinander folgen.

Record-Typen

werden als kontinuierliche Folge ihrer Felder gespeichert, wobei das bei der Definition zuerst aufgeführte Feld auf der niedrigsten Adresse steht. Variante Teile belegen jeweils denselben Speicherbereich, wobei auch hier in der Reihenfolge der einzelnen Komponenten gearbeitet wird.

Datei-Typen

werden intern als Records dargestellt. Typisierte und untypisierte Dateien belegen jeweils 128 Bytes Speicherplatz, die folgendermaßen strukturiert sind:

type

{ typisierte und untypisierte Dateien }

FileRec = **record**

Handle: Word;

Mode: Word;

RecSize: Word;

Private: **array**[1..26] **of** Byte;

UserData: **array**[1..16] **of** Byte;

Name: **array**[0..79] **of** Char;

end;

Textdateien belegen jeweils 256 Bytes und haben das folgende Format:

type

```

{ Textdateien }
TextBuf = array[0..127] of Char;
TextRec = record
    Handle:    Word;
    Mode:      Word;
    BufSize:   Word;
    Private:   Word;
    BufPos:    Word;
    BufEnd:    Word;
    BufPtr:    ^TextBuf;
    OpenFunc:  Pointer;
    InOutFunc: Pointer;
    FlushFunc: Pointer;
    CloseFunc: Pointer;
    UserData:  array[1..16] of Byte;
    Name:      array[0..79] of Char;
    Buffer:     TextBuf;
end;

```

Wenn die Datei offen ist, enthält das Feld *Handle* die von DOS vergebene Datei-Referenznummer. Das Feld *Mode* enthält einen der folgenden »magischen« Werte (die *nicht* von 0..3 durchnummeriert sind):

const

```

fmClosed  = $D7B0; { Datei ist geschlossen }
fmInput    = $D7B1; { nur Lesen }
fmOutput   = $D7B2; { nur Schreiben }
fmInOut    = $D7B3; { Lesen und Schreiben erlaubt }

```

Der Wert *fmInOut* ist für Textdateien nicht möglich – nur typisierte und untypisierte Dateien können gleichzeitig gelesen und beschrieben werden. Alle anderen Werte bedeuten, daß die entsprechende Datei nicht offen ist – *Mode* ist in diesen Fällen nicht weiter definiert.

Das Feld *UserData* wird von Turbo Pascal und DOS nicht benutzt und steht dem Programmierer für eigene Zwecke zur Verfügung.

Das Feld *Name* enthält den Namen der Datei – allerdings nicht als String mit einem vorangestellten Längenbyte, sondern im Format ASCIIZ: einer beliebigen Folge von Zeichen, die mit einem NUL (ASCII-Code 0) abgeschlossen ist.

Typisierte und untypisierte Dateien enthalten im Feld *RecSize* die Recordlänge (d.h. die Größe einer Komponente); das Feld *Private* wird zwar nicht direkt verwendet, ist aber »reserviert«.

Textdateien enthalten im Feld *BufPtr* einen Zeiger auf einen Puffer mit der durch *BufSize* in Bytes angegebenen Größe. *BufPos* enthält den Index des nächsten zu schreibenden bzw. zu lesenden Zeichens innerhalb dieses Puffers, *BufEnd* die Gesamtzahl der momentan im Puffer gespeicherten Zeichen. Die Zeiger *OpenFunc*, *InOutFunc*, *FlushFunc* und *CloseFunc* zeigen auf die Routine, über die die jeweilige Funktion (*Open*, *Close* usw.) ausgeführt wird. Der Abschnitt »Text-Gerätetreiber« in diesem Kapitel setzt sich näher damit auseinander.

Prozedurale Typen

speichern die Adresse der ihr zugeordneten Routinen mit zwei aufeinanderfolgenden Wörtern (Datentyp *Word*): auf der niedrigeren Adresse steht der Offset-Anteil, auf der höheren Adresse der Segment-Anteil.

Direkte Zugriffe auf Speicher-Adressen

Turbo Pascal stellt drei vordefinierte Arrays zur Verfügung, die direkte Zugriffe auf einzelne Speicheradressen und Ports (I/O-Adressen) ermöglichen:

Die Arrays *Mem*, *MemW* und *MemL*

Mem ist als *array of Byte*, *MemW* als *array of Word* und *MemL* als *array of LongInt* definiert. Alle drei haben eine spezielle Notation ihrer Indizes gemeinsam: zwei Ausdrücke des Typs *Word*, die durch einen Doppelpunkt voneinander getrennt sind. Einige Beispiele:

```
Mem[$0040: $0049] := 7;
Wert := MemW[Seg(V) : OfS(V)];
LangWert := MemL[64: 3*4];
```

Der erste Befehl setzt die Speicherzelle mit der Adresse \$0040:\$0047 auf den Wert 7, der zweite liest die ersten beiden Bytes der Variablen *V* und ordnet sie der Integer-Variablen *Wert* zu – falls es sich bei *V* ebenfalls um einen Integer-Typ handelt, ist er mit dem Befehl *Wert := V* identisch. Der dritte Befehl schließlich liest die Speicherzellen \$0040:\$000C..\$0040:\$000F und ordnet ihren Wert der Variablen *LangWert* zu, die den Typ *LongInt* haben muß.

Das interne Datenformat von Objekten

Das interne Datenformat eines Objekts ist dem eines Records sehr ähnlich. Die Felder eines Objekts werden in der Reihenfolge der Deklaration als eine zusammenhängende Folge von Variablen gespeichert. Jedes Feld, das von einem Vorfahren vererbt wird, wird vor den Feldern gespeichert, die vom Nachkommen neu definiert werden.

Wenn ein Objekttyp virtuelle Methoden, Konstruktoren oder Destruktoren definiert, dann reserviert der Compiler ein zusätzliches Feld im Objekttyp. Dieses 16-Bit-Feld, das als *Feld der virtuellen Methodentabelle* (VMT-Feld) bezeichnet wird, dient dazu, den Offset der VMT der entsprechenden Methode im Datensegment zu speichern. Das VMT-Feld steht direkt hinter den normalen Feldern des Objekttyps. Wenn ein Objekttyp virtuelle Methoden, Konstruktoren

oder Destruktoren erbt, erbt er auch das VMT-Feld, so daß kein zusätzliches VMT-Feld reserviert wird.

Initialisiert wird das VMT-Feld einer Instanz von den Konstruktoren des Objekttyps. Ein Programm initialisiert das VMT-Feld niemals selbst und greift auch nicht auf das VMT-Feld zu.

Die folgenden Beispiele veranschaulichen die internen Datenformate von Objekttypen.

type

```

LocationPtr = ^Location;
Location = object
  X, Y: Integer;
  procedure Init (PX, PY: Integer);
  function GetX: Integer;
  function GetY: Integer;
end;

PointPtr = ^Point;
Point = object(Location)
  Color: Integer;
  constructor Init (PX, PY, PColor: Integer);
  destructor Done; virtual;
  procedure Show; virtual;
  procedure Hide; virtual;
  procedure MoveTo(PX, PY: Integer); virtual;
end;

CirclePtr = ^Circle;
Circle = object(Circle)
  Radius: Integer;
  constructor Init(PX, PY, PColor, PRadius: Integer);
  procedure Show; virtual;
  procedure Hide; virtual;
  procedure Fill; virtual;
end;

```

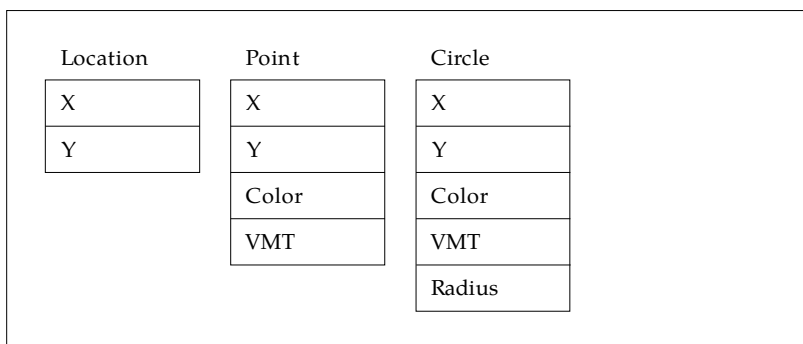


Abbildung 14.12 Aufbau der Instanzen von Location, Point und Circle.

Abbildung 14.12 enthält den Aufbau der Instanzen von *Location*, *Point* und *Circle*. Jedes Kästchen stellt eine Speicherstelle dar.

Da *Point* in der Hierarchie der erste Datentyp mit virtuellen Methoden ist, folgt das VMT-Feld direkt hinter dem Feld *Color*.

***Virtuelle
Methodentabellen***

Jeder Objekttyp, der virtuelle Methoden, Konstruktoren oder Destruktoren enthält oder erbt, ist mit einer Tabelle virtueller Methoden (VMT) verbunden, die im initialisierten Abschnitt des Datensegments gespeichert wird. Es gibt nur eine VMT pro Objekttyp (und nicht für jede einzelne Instanz), aber zwei verschiedene Objekttypen können eine VMT niemals gemeinsam verwenden, gleichgültig wie ähnlich sie sich auch sein mögen. VMTs werden vom Compiler automatisch erstellt. Sie dürfen auch niemals direkt von einem Programm aus verändert werden. Genauso werden die Zeiger auf die VMTs automatisch in den Instanzen des Objekttyps vom Konstruktor des Objekttyps gespeichert und dürfen von einem Programm niemals direkt manipuliert werden.

Das erste Word einer VMT enthält die Größe der Instanzen des damit verbundenen Objekttyps. Diese Information wird von anderen Konstruktoren und Destruktoren verwendet, um festzustellen, wieviele Bytes über die erweiterte Syntax der Standardprozeduren *New* und *Dispose* reserviert oder freigegeben werden sollen:

Point VMT	Circle VMT
\$0008	\$0008
\$FFF8	\$FFF8
@Point.Done	@Point.Done
@Point.Show	@Point.Show
@Point.Hide	@Point.Hide
@Point.MoveTo	@Point.MoveTo
	@Circle.Fill

Abbildung 14.13 Der Aufbau der VMTs von *Point* und *Circle*

Das zweite Word einer VMT enthält die negativen Größen der Instanzen des damit verbundenen Objekttyps. Diese Information wird mit Hilfe eines Überprüfungsmechanismus während des Aufrufs von Methoden verwendet, um nicht initialisierte Objekte (Instanzen, die ohne einen Aufruf eines Konstruk-

tors, vorgenommen wurden) und die Konsistenz der VMT zu prüfen. Wenn die Überprüfung der Aufrufe virtueller Funktionen (mit $\{R+\}$) eingeschaltet ist, dann erzeugt der Compiler einen Aufruf der VMT-Überprüfungsfunktion vor jedem Aufruf einer virtuellen Funktion. Die VMT-Überprüfungsfunktion schaut nach, ob das erste Word einer VMT ungleich Null ist, und ob die Summe des ersten und des zweiten Words Null ergibt. Wenn eine dieser Bedingungen nicht zutrifft, wird der Laufzeitfehler 210 erzeugt.

Achtung: Das Einschalten der Bereichsüberprüfung und die Überprüfung des Aufrufs virtueller Funktionen verlangsamt Ihr Programm. Das Programm wird etwas größer, so daß Sie $\{R+\}$ nur beim Debuggen einschalten und beim fertigen Programm auf $\{R-\}$ umschalten sollten.

Schließlich kommt beim Offset 4 in der VMT eine Liste von 32-Bit-Zeigern zu den entsprechenden Methoden, ein Zeiger für jede Methode im Objekttyp in der Reihenfolge der Deklaration. Jeder Eintrag enthält die Adresse des entsprechenden Einsprungpunkts der virtuellen Methode.

Abbildung 5.2 zeigt den Aufbau der VMTs der Datentypen *Point* und *Circle* (der Datentyp *Location* hat keine VMT, da er keine virtuellen Methoden, Konstruktoren oder Destruktoren enthält). Jedes kleine Kästchen entspricht einem Word im Speicher und jedes größere Kästchen entspricht zwei Words.

Achten Sie darauf, wie *Circle* die Methoden *Done* und *MoveTo* von *Point* erbt und die Methoden *Show* und *Hide* ersetzt.

Wie bereits früher erwähnt wurde, enthalten die Konstruktoren eines Objekttyps einen speziellen Programmcode, der den Offset der VMT des Objekts in der zu initialisierenden Instanz ablegt. Wenn zum Beispiel eine Instanz *P* eines Punkts und die Instanz *C* eines Kreises vorliegt, dann wird bei einem Aufruf von *P.Init* automatisch der Offset der VMT von *Point* im VMT-Feld von *P* gespeichert und bei einem Aufruf von *C.Init* in gleicher Weise der Offset der VMT von *Circle* im VMT-Feld von *C* gespeichert. Diese automatische Initialisierung ist ein Teil des Codes, der vom Konstruktor angesprungen wird, so daß das VMT-Feld *Self* bereits richtig gesetzt ist, wenn das Programm zum ersten **begin** des Anweisungsteils im Konstruktor kommt. Ein Konstruktor kann also im Bedarfsfall die virtuellen Methoden aufrufen.

Die Standardfunktion SizeOf

Wenn die Standardfunktion *SizeOf* auf eine Instanz eines Objekttyp mit einer VMT angewendet wird, dann liefert *SizeOf* die Größe, die in der VMT gespeichert ist. Für Objekttypen, die eine VMT haben, erhält man also mit *SizeOf* die tatsächliche Größe der Instanz und nicht die deklarierte Größe.

Die Standardfunktion TypeOf

Turbo Pascal definiert eine Standardfunktion, *TypeOf*, die einen Zeiger auf die VMT eines Objekttyps liefert. *TypeOf* erhält einen einzigen Parameter. Das

kann entweder die Bezeichnung eines Objekttyps oder eine Instanz eines Objekttyps sein. In beiden Fällen ist das Ergebnis vom Typ *Pointer* ein Zeiger auf die VMT des Objekttyps. *TypeOf* darf nur auf Objekttypen mit einer VMT angewendet werden. Bei allen anderen Datentypen liefert die Funktion einen Fehler. Die Funktion *TypeOf* kann man zum Überprüfen des tatsächlichen Datentyps einer Instanz verwenden. Zum Beispiel:

```
if TypeOf(Self) = TypeOf(Point) then ...
```

Der Aufruf von virtuellen Methoden

Um eine virtuelle Methode aufzurufen, erzeugt der Compiler Code, um die VMT-Adresse aus dem VMT-Feld im Objekt zu lesen, und führt dann den Aufruf der Methode über den zugehörigen Eintrag aus. Mit einer Variable *PP* des Typs *PointPtr* erzeugt der Aufruf *PP^.Show* zum Beispiel den folgenden Code:

```
les    di,PP                ; PP in ES:DI laden
push   es                  ; als Self-Parameter übergeben
push   di
mov     di,es:[di+6]        ; VMT-Offset aus dem VMT-Feld holen
call    DWORD PTR [di+8]    ; VMT-Einsprungspunkt für Show aufrufen
```

Die Kompatibilitätsregeln für Objekttypen erlauben, daß *PP* auf *Point* oder *Circle* oder einen anderen Nachkommen von *Point* zeigt. Und wenn Sie die hier dargestellten VMTs untersuchen, dann sehen Sie, daß bei *Point* der Einsprungspunkt beim Offset 8 in der VMT auf *Point.Show* zeigt, während er bei *Circle* auf *Circle.Show* zeigt. Je nachdem, welcher Datentyp von *PP* zur Laufzeit tatsächlich vorliegt, ruft der *CALL*-Befehl entweder *Point.Show* oder *Circle.Show*, oder die Methode *Show* eines anderen Nachkommens von *Point* auf.

Wenn *Show* eine statische Methode gewesen wäre, dann wäre für *PP.Show* folgender Code erzeugt worden:

```
les    di,PP                ; PP in ES:DI laden
push   es                  ; als Self-Parameter übergeben
push   di
call    Point.Show          ; direkter Aufruf von Point.Show
```

Hier spielt es keine Rolle, wohin *PP* zeigt, im Code wird immer die Methode *Point.Show* aufgerufen.

Aufrufkonventionen von Methoden

Methoden verwenden dieselben Aufrufkonventionen wie normale Prozeduren und Funktionen, außer daß jede Methode einen zusätzlichen Parameter, *Self*, implizit übergibt, der einem *var*-Parameter mit dem Datentyp der Methode

entspricht. Der *Self*-Parameter wird immer als letzter Parameter in Form eines 32-Bit-Zeigers auf die Instanz übergeben, über die die Methode aufgerufen wurde. Wenn zum Beispiel eine Variable *PP* des Typs *PointPtr* definiert ist, dann wird der Aufruf *PP^.MoveTo(10,20)* folgendermaßen codiert:

```

mov    ax,10                                ; 10 in AX laden
push   ax                                  ; als PX-Parameter übergeben
mov    ax,20                                ; 20 in AX laden
push   ax                                  ; als PY-Parameter übergeben
les    di,PP                                ; PP in ES:DI laden
push   es                                  ; als Self-Parameter übergeben
push   di
mov    di,es:[di+6]                          ; VMT-Offset aus VMT-Feld laden
call   DWORD PTR [di+16]                    ; VMT-Eintrag für MoveTo aufrufen

```

Bei der Rückkehr muß die Methode den *Self*-Parameter vom Stack holen, wie dies bei allen anderen Parametern auch der Fall ist.

Die Methoden verwenden immer FAR-Aufrufe, egal, wie die Compileranweisung *\$F* eingestellt ist.

Konstruktoren und Destruktoren

Konstruktoren und Destruktoren verwenden dieselben Aufrufkonventionen wie normale Methoden, mit Ausnahme eines zusätzlichen Parameters, der ein Word umfaßt. Dies ist der *VMT*-Parameter, der direkt vor dem *Self*-Parameter auf den Stack gelegt wird.

Bei den Konstruktoren enthält der *VMT*-Parameter den *VMT*-Offset, um das *VMT*-Feld des *Self*-Parameters bei der Initialisierung des *Self*-Parameters zu speichern. Außerdem wird ein **nil**-Zeiger im *Self*-Parameter übergeben, wenn ein Konstruktor zum Reservieren eines dynamischen Objekts mit Hilfe der erweiterten Syntax der Standardprozedur *New* aufgerufen wird. Deshalb muß der Konstruktor ein neues dynamisches Objekt anlegen, dessen Adresse bei der Rückkehr des Konstruktors in *DX:AX* an das aufrufende Programm zurückgegeben wird. Wenn der Konstruktor das Objekt nicht anlegen kann, wird in *DX:AX* ein **nil**-Zeiger zurückgegeben (siehe den Abschnitt »Fehlerbehandlung in Konstruktoren«).

Wenn schließlich ein Konstruktor mit einem vollständigen Methodenbezeichner aufgerufen wird (d.h. zusammen mit einer Bezeichnung des Objekttyps, der ein Punkt und der Methodenbezeichner folgen), wird der Wert Null als Parameter in die *VMT*-Tabelle eingetragen. Dies zeigt dem Konstruktor an, daß das *VMT*-Feld von *Self* nicht initialisiert werden soll.

Für einen Destruktor gibt eine Null als *VMT*-Parameter einen normalen Aufruf an. Ein von Null verschiedener Wert zeigt an, daß der Destruktor mit der erweiterten Syntax der Standardprozedur *Dispose* aufgerufen wurde. Dies führt beim Destruktor dazu, daß *Self* vor der Rückkehr freigegeben wird (die Größe von *Self* finden Sie im ersten Wort der *VMT* von *Self*).

Erweiterungen von New und Dispose

Die Standardprozeduren *New* und *Dispose* (siehe Kapitel 1 »Verzeichnis der Routinen« des *Referenzhandbuchs*) wurden erweitert, damit man einen Aufruf eines Konstruktors bzw. Destruktors als zweiten Parameter zum Anlegen oder Freigeben einer dynamischen Objektvariable optional angeben kann. Die Syntax lautet:

```
New(P, Konstrukt)
```

und

```
Dispose(P, Destrukt)
```

wobei *P* eine Zeigervariable ist, die auf einen Objekttyp zeigt, und *Konstrukt* und *Destrukt* Aufrufe von Konstruktoren und Destruktoren eines Objekttyps sind. Bei *New* bedeutet die neue Syntax nichts anderes als:

```
New(P);  
P^.Konstrukt;
```

und für *Dispose* ist die erweiterte Syntax das gleiche wie die Ausführung der Anweisungen

```
P^.Destrukt;  
FreeMem(P, SizeOf(P^));
```

Ohne die Syntaxerweiterung würden solche »Paare« von Aufrufen von *New* gefolgt von einem Konstruktor und einem Destruktor, hinter denen wiederum ein *Dispose* steht, sehr häufig vorkommen. Die erweiterte Syntax verbessert die Lesbarkeit und erzeugt außerdem einen kürzeren und effektiveren Code.

Der nachfolgende Ausschnitt verdeutlicht die Anwendung der erweiterten Syntax von *New* und *Dispose*:

```
var  
  SP : StrFieldPtr;  
  ZP : ZipFieldPtr;  
begin  
  New(SP, Init(1, 1, 25, 'Vorname'));  
  New(ZP, Init(1, 2, 5, 'ZIP', 0, 99999));  
  SP^.Edit;  
  ZP^.Edit;  
  ...  
  Dispose(ZP, Done);  
  Dispose(SP, Done);  
end;
```

Eine zusätzliche Erweiterung ermöglicht die Verwendung von *New* als *Funktion*, die eine dynamische Variable eines bestimmten Typs anlegt und einen Zeiger liefert. Die Syntax lautet dann:

```
New(T)
```

oder

```
New(T, Konstrukt)
```

In der ersten Form kann *T* ein beliebiger Zeigertyp sein. In der zweiten Form muß *T* auf einen Objekttyp zeigen. *Konstrukt* muß ein Aufruf eines Konstruktors des entsprechenden Objekttyps sein. In beiden Fällen ist der Typ des Funktionsresultats *T*. Hier ein Beispiel:

```
var
  F1, F2 : FieldPtr;
begin
  F1 := New(StrFieldPtr, Init(1, 1, 25, 'Vorname'));
  F2 := New(ZIPFieldPtr, Init(1, 2, 5, 'ZIP', 0, 99999));
  ...
  WriteLn(F1^.GetStr);      { Aufruf von StrField.GetStr }
  WriteLn(F2^.GetStr);      { Aufruf von ZIPField.GetStr }
  ...
  Dispose(F2, Done);        { Aufruf von Field.Done }
  Dispose(F1, Done);        { Aufruf von StrField.Done }
end;
```

Beachten Sie, daß *F1* und *F2*, auch wenn sie den Typ *FieldPtr* haben, über die erweiterte Zuweisungskompatibilität von Zeigern einen Zeiger erhalten können, der auf einen Nachkommen von *Field* zeigt. Da *GetStr* und *Done* virtuelle Methoden sind, wird der Auswahlmechanismus für virtuelle Methoden die richtigen Funktionen *StrField.GetStr*, *ZipField.GetStr*, *Field.Done* sowie auch *StrField.Done* aufrufen.

Methoden in Assembler

Methoden, die in Assembler implementiert wurden, können mit dem integrierten Assembler oder mit Hilfe der Compiler-Anweisung *\$L* und des Schlüsselworts **external** in Turbo Pascal eingebunden werden. Die Deklaration einer externen Methode in einem Objekttyp unterscheidet sich nicht von einer normalen Methode. Die Implementation dieser Methode enthält nur den Deklarationskopf der Methode, dem das reservierte Wort **external** folgt.

In einem Assembler-Quelltext wird das Zeichen @ anstelle des Punkts (.) für einen vollständigen Bezeichner angegeben (der Punkt hat nämlich in Assembler bereits eine andere Bedeutung und darf deshalb nicht in einem Variablenbezeichner verwendet werden). Der Pascal-Bezeichner *Rect.Init* wird zum Beispiel in Assembler als *Rect@Init* geschrieben. Die Syntax mit dem @-Zeichen kann man sowohl in *PUBLIC*- als auch in *EXTRN*-Variablenbezeichnern verwenden.

Als Beispiel für eine Methode in Assembler haben wir ein einfaches *Rect*-Objekt implementiert.

```
type
```

```

Rect = object
  X1, Y1, X2, Y2: Integer;
  procedure Init(XA, YA, XB, YB: Integer);
  procedure Union(VAR R: Rect);
  function Contains(X, Y: Integer): Boolean;
end;

```

Ein Objekt *Rect* ist ein Rechteck, das mit Hilfe der vier Koordinaten, *X1*, *Y1*, *X2* und *Y2* dargestellt wird. Die linke obere Ecke des Rechtecks ist durch *X1* und *Y1* festgelegt und die rechte untere Ecke ist durch *X2* und *Y2* definiert. In der Methode *Init* werden den Rechteckkoordinaten Werte zugewiesen. Die Methode *Union* berechnet das kleinste Rechteck, das sowohl das Rechteck selbst als auch ein anderes Rechteck enthält. In der Methode *Contains* wird der Wert *True* zurückgegeben, wenn der angegebene Punkt innerhalb des Rechtecks liegt, und der Wert *False*, wenn das nicht der Fall ist. Andere Methoden, zum Beispiel Verschieben, Verändern der Größe, Berechnen von Schnittflächen und Vergleiche mit anderen Rechtecken, die *Rect* zu einem brauchbaren Objekt komplettieren, lassen sich leicht implementieren.

Die Pascal-Implementationen der Methoden von *Rect* enthalten nur den Deklarationskopf mit dem Schlüsselwort **external**.

```

{$L RECT}

procedure Rect.Init(XA, YA, XB, YB: Integer); external;
procedure Rect.Union(var R: Rect); external;
procedure Rect.Contains(X, Y: Integer): Boolean; external;

```

Es ist natürlich keineswegs erforderlich, daß alle Methoden als externe Unterprogramme deklariert werden. Jede der einzelnen Methoden kann entsprechend den Erfordernissen entweder in Pascal oder in Assembler implementiert werden.

Die Assembler-Quelldatei RECT.ASM, in der die drei externen Methoden implementiert sind, ist nachfolgend abgedruckt:

```

                TITLE      Rect
                LOCALS     @@
; Rechteck-Struktur
Rect           STRUC
X1             DW         ?
Y1             DW         ?
X2             DW         ?
Y2             DW         ?
Rect           ENDS
code           SEGMENT    BYTE PUBLIC
                ASSUME     cs:code
; procedure Rect.Init(XA, YA, XB, YB)
                PUBLIC     Rect@Init
Rect@Init      PROC       FAR
@XA            EQU        (WORD PTR [bp+16])
@YA            EQU        (WORD PTR [bp+14])

```

```

@XB      EQU      (WORD PTR [bp+12])
@YB      EQU      (WORD PTR [bp+10])
@Self    EQU      (DWORD PTR [bp+6])

    push    bp      ; BP sichern
    mov     bp,sp    ; Stackaufbau festlegen
    les     di,@Self ; Self in ES:DI laden
    cld                     ; Zeiger nach oben
    mov     ax,@XA    ; X1 := XA
    stosw
    mov     ax,@YA    ; Y1 := YA
    stosw
    mov     ax,@XB    ; X2 := XB
    stosw
    mov     ax,@YB    ; YB := YB
    stosw
    pop     bp      ; BP wiederherstellen
    ret     12      ; Parameter abräumen, Rückkehr

Rect@Init      ENDP

; procedure Rect.Union(var R: Rect)
      PUBLIC   Rechteck@Union
Rect@Union      PROC      FAR
@R      EQU      (DWORD PTR [bp+10])
@Self    EQU      (DWORD PTR [bp+6])

    push    bp      ; BP sichern
    mov     bp,sp    ; Stackaufbau festlegen
    push    ds      ; DS sichern
    lds     si,@R    ; R in DS:SI laden
    les     di,@Self ; Self in ES:DI laden
    cld                     ; Zeiger nach oben
    lodsw                      ; Falls R.X1 >= X1 dann @@1
    scasw
    jge     @@1
    dec     di      ; X1 := R.X1
    dec     di
    stosw
@@1:    lodsw                      ; falls R.Y1 >= Y1 dann @@2
    scasw
    jge     @@2
    dec     di      ; Y1 := R.Y1
    dec     di
    stosw
@@2:    lodsw                      ; falls R.X2 <= X2 dann @@3
    scasw
    jle     @@3
    dec     di      ; X2 := R.X2
    dec     di
    stosw
@@3:    lodsw                      ; falls R.Y2 <= Y2 dann @@4

```

```

        scasw
        jle     @@4
        dec     di          ; Y2 := R.Y2
        dec     di
        stosw
@@4:    pop     ds          ; DS wiederherstellen
        pop     bp          ; BP wiederherstellen
        ret     8          ; Parameter abräumen, Rückkehr
Rect@Union      ENDP
; function Rect@Contains(X, Y: Integer): Boolean
        PUBLIC  Rect@Contains
Rect@Contains    PROC      FAR
@@X             EQU       (WORD PTR [bp+12])
@@Y             EQU       (WORD PTR [bp+10])
@@Self          EQU       (DWORD PTR [bp+6])
        push    bp          ; BP sichern
        mov     bp,sp        ; Stackaufbau festlegen
        les     di,@Self    ; Self in ES:DI laden
        mov     al,0        ; False zurückgeben
        mov     dx,@X       ; if (X < X1) or (X > X2) goto @@1
        cmp     dx,es:[di].X1
        jl      @@1
        cmp     dx,es:[di].X2
        jg      @@1
        mov     dx,@Y       ; if (Y < Y1) or (Y > Y2) goto @@1
        cmp     dx,es:[di].Y1
        jl      @@1
        cmp     dx,es:[di].Y2
        jg      @@1
        inc     ax
@@1:    pop     bp          ; BP wiederherstellen
        ret     8          ; Parameter abräumen, Rückkehr
Rect@Contains    ENDP
code            ENDS
END

```

Fehlerbehandlung in Konstruktoren

Wie zu Beginn dieses Kapitels im Abschnitt »Die Variable *HeapError*« beschrieben, können Sie eine Fehlerbehandlungsfunktion für Probleme am Heap mit Hilfe der Variable *HeapError* in der Unit *System* installieren. Diese Möglichkeit wird auch in Turbo Pascal 6.0 unterstützt, aber nun wird außerdem noch der Programmablauf in einem Konstruktor beeinflusst.

Standardmäßig wird der Laufzeitfehler 203 erzeugt, wenn bei der Speicherplatzreservierung einer dynamischen Objektinstanz, die mit Hilfe der erweiter-

ten Syntax von *New* vorgenommen wird, nicht mehr genug Speicherplatz vorhanden ist. Wenn Sie eine Heap-Fehlerfunktion installieren, die anstelle des standardmäßigen Funktionsergebnisses von 0 den Wert 1 zurückgibt, dann liefert ein Aufruf eines Konstruktors mittels *New* den Wert **nil**, wenn die Anforderung nicht erfüllt werden kann (statt das Programm abzurechnen).

Der Code zur Speicherplatzreservierung und Initialisierung des VMT-Felds einer dynamischen Instanz ist Teil der Einsprungssequenz eines Konstruktors: Am **begin** des Anweisungsteils des Konstruktors ist die Instanz bereits angelegt und initialisiert. Scheitert die Speicherplatzreservierung und wird von der Heap-Fehlerfunktion eine 1 zurückgegeben, dann übergeht der Konstruktor die Ausführung des Anweisungsteils und gibt einen **nil**-Zeiger zurück. Der in der *New*-Anweisung angegebene Zeiger, mit dem der Konstruktor aufgerufen wurde, wird also auf **nil** gesetzt.

Ist das Programm am **begin** des Anweisungsteils des Konstruktors angelangt, so wurde die Instanz des Objekttyps erfolgreich angelegt und initialisiert. Wenn jedoch der Konstruktor selbst dynamische Variablen anzulegen versucht, um zum Beispiel die Zeigerfelder in einer Instanz zu initialisieren, kann es ebenfalls vorkommen, daß die Reservierung dieses Speicherplatzes scheitert. Wenn das passiert, sollte ein sauber konzipierter Konstruktor die vorgenommenen Reservierungen wieder rückgängig machen und die Instanz des Objekttyps freigeben, so daß ein **nil**-Zeiger zurückgegeben werden kann. Damit dieser »Rückzug« möglich ist, wurde in Turbo Pascal eine neue Standardprozedur, *Fail*, implementiert, die keine Parameter hat und nur von einem Konstruktor aus aufgerufen werden darf. Ein Aufruf von *Fail* führt dazu, daß ein Konstruktor die dynamische Instanz, die vorher beim Einsprung in den Konstruktor angelegt wurde, wieder freigibt, und daß sofort die Rückkehr mit dem Ergebnis **nil** angetreten wird, um den Fehler anzuzeigen.

Wenn dynamische Instanzen mit Hilfe der erweiterten Syntax von *New* angelegt werden, zeigt das Ergebnis **nil** in der entsprechenden Zeigervariable an, daß die Operation gescheitert ist. Leider gibt es keine solche Zeigervariable, die man nach der Erstellung einer statischen Instanz oder dem Aufruf eines vererbten Konstruktors untersuchen könnte. Stattdessen erlaubt Turbo Pascal den Einsatz eines Konstruktors als Boolean-Funktion in einem Ausdruck: Das Ergebnis *True* steht für eine erfolgreiche Durchführung und *False* zeigt an, daß der Vorgang gescheitert ist, weil die Prozedur *Fail* innerhalb eines Konstruktors aufgerufen werden mußte.

Im folgenden Programm sind zwei einfache Objekttypen mit Zeigern implementiert. Die erste Version des Programms enthält keine Fehlerbehandlung im Konstruktor.

type

```
LinePtr = ^Line;
Line = string[79];
BasePtr = ^Base;
Base = object
  L1, L2: LinePtr;
  constructor Init(S1, S2: Line);
  destructor Done; virtual;
```

```

    procedure Dump; virtual;
end;

DerivedPtr = ^Derived;
Derived = object (Base)
    L3, L4: LinePtr;
    constructor Init(S1, S2, S3, S4: Line);
    destructor Done; virtual;
    procedure Dump; virtual;
end;

var
    BP: BasePtr;
    DP: DerivedPtr;

constructor Base.Init(S1, S2: Line);
begin
    New(L1);
    New(L2);
    L1^ := S1;
    L2^ := S2;
end;

destructor Base.Done;
begin
    Dispose(L2);
    Dispose(L1);
end;

procedure Base.Dump;
begin
    WriteLn('B: ', L1^, ' ', L2^, '.');
end;

constructor Derived.Init(S1, S2, S3, S4: Line);
begin
    Base.Init(S1, S2);
    New(L3);
    New(L4);
    L3^ := S3;
    L4^ := S4;
end;

destructor Derived.Done;
begin
    Dispose(L4);
    Dispose(L3);
    Base.Done;
end;

procedure Derived.Dump;
begin

```

```

    WriteLn('A: ', L1^, ', ', L2^, ', ', L3^, ', ', L4^, '.');
end;

begin
    New(BP, Init('Turbo', 'Pascal'));
    New(DP, Init('Nord', 'Ost', 'Süd', 'West' ));
    BP^.Dump;
    DP^.Dump;
    Dispose(DP, Done);
    Dispose(BP, Done);
end.

```

Das nächste Beispiel zeigt, wie das vorherige Beispiel mit einer Fehlerbehandlungsroutine umgeschrieben werden kann.

Die Typen- und Variablendeklarationen bleiben gleich, deshalb werden sie nicht mehr wiederholt.

```

constructor Base.Init(S1, S2: Line);
begin
    New(L1);
    New(L2);
    if (L1 = nil) or (L2 = nil) then
        begin
            Base.Done;
            Fail;
        end;
    L1^ := S1;
    L2^ := S2;
end;

destructor Base.Done;
begin
    if L2 <> nil then Dispose(L2);
    if L1 <> nil then Dispose(L1);
end;

constructor Derived.Init(S1, S2, S3, S4: Line);
begin
    if not Base.Init(S1, S2) then Fail;
    New(L3);
    New(L4);
    if (L3 = nil) or (L4 = nil) then
        begin
            Derived.Done;
            Fail;
        end;
    L3^ := S3;
    L4^ := S4;
end;

destructor Derived.Done;

```

```

begin
  if L4 <> nil then Dispose(L4);
  if L3 <> nil then Dispose(L3);
  Base.Done;
end;

function HeapFunc(Groesse: Word): Integer; far;
begin
  HeapFunc := 1;
end;

begin
  HeapError := @HeapFunc;      { Heap-Fehlerbehandlung installieren }
  New(BP, Init('Turbo', 'Pascal'));
  New(DP, Init('Nord', 'Ost', 'Süd', 'West' ));
  if (BP = nil) or (DP = nil) then
    WriteLn('Fehler bei Speicherplatzreservierung');
  else
    begin
      BP^.Dump;
      DP^.Dump;
    end;
  if DP <> nil then Dispose(DP, Done);
  if BP <> nil then Dispose(BP, Done);
end.

```

Achten Sie darauf, wie die entsprechenden Destruktoren in *Base.Init* und *Derived.Init* eingesetzt werden, um alle vorher vorgenommenen Speicherplatzreservierungen zurückzunehmen, bevor schließlich *Fail* aufgerufen wird, um den Vorgang endgültig abzubrechen. Außerdem sollten Sie beachten, daß der Aufruf von *Base.Init* in *Derived.Init* innerhalb eines Ausdrucks codiert wurde, damit der erfolgreiche Abschluß des vererbten Konstruktors überprüft werden kann.

Der Aufruf von Funktionen und Prozeduren

Die Übergabe von Parametern beim Aufruf einer Prozedur oder Funktion geschieht durch den Stack des Prozessors: Die Parameter werden in der Reihenfolge auf den Stack gelegt, die durch ihre Deklaration festgelegt ist, und innerhalb der aufgerufenen Routine wieder vom Stack gelesen bzw. entfernt. Im Prinzip lassen sich die Maschinenbefehle für einen Aufruf wie

```
BeispielProc(X, Y, Z);
```

also folgendermaßen darstellen:

```
PUSH <X>
PUSH <Y>
```

```
PUSH <Z>  
CALL <BeispielProc>
```

Parameter werden entweder als Wert oder als Variablen-Referenz übergeben. Bei einer Wertübergabe wird lediglich der entsprechende Wert auf den Stack gelegt, bei einer Variablen-Übergabe dagegen die Adresse der entsprechenden Variablen.

Variablen- Parameter

werden dann verwendet, wenn die aufgerufene Routine den entsprechenden formalen Parameter als **var** deklariert. Die aufgerufene Routine bekommt einen Zeiger übergeben: sie liest den tatsächlichen Wert selbst, hat also direkten Zugriff auf die entsprechende Variable und kann ihren Wert verändern. Der übergebene Zeiger entspricht in jedem Fall dem Datentyp *Pointer* (4 Bytes, organisiert als 2 Werte des Typs *Word*).

Wert- Parameter

werden – je nach Größe und Typ – entweder als Wert oder durch einen Zeiger übergeben. Abgesehen von Gleitkommazahlen, bei denen Turbo Pascal (zum einen aus »historischen«, zum anderen aus Kompatibilitätsgründen) ebenfalls eine Wertübergabe verwendet, werden Daten mit 1, 2 oder 4 Bytes Umfang als Werte und alle »größer« Typen als Zeiger übergeben. Im letzteren Fall kann die Original-Variable trotzdem innerhalb der aufgerufenen Routine nicht verändert werden: Turbo Pascal stellt automatisch einige Befehle voran, die die tatsächlichen Werte in einen lokalen Speicherbereich kopieren, und »vergißt« den übergebenen Zeiger danach.

Hinweis: Da die Prozessoren der 8086-Familie nicht über PUSH- und POP-Befehle für einzelne Bytes verfügen, werden Parameter mit einem Byte ebenfalls als Wort (2 Bytes) übergeben. Der tatsächliche Wert befindet sich in diesem Fall im niederwertigen Byte, das höherwertige Byte ist unbenutzt (und undefiniert).

- Integer-Typen werden im Format ihrer Speicherung übergeben, d.h. je nach Typ mit einem, zwei oder vier Bytes. Für Werte des Typs *LongInt* legt Turbo Pascal das niederwertige Wort zuletzt auf den Stack, das dadurch auch auf der niedrigeren Adresse innerhalb des Stack-Segments steht.
- Char-Typen werden in derselben Weise übergeben wie *Byte*, d.h. als einzelnes und vorzeichenloses Byte.
- Boolesche Werte werden als einzelnes Byte mit dem Wert 0 oder 1 übergeben.
- Aufzählungstypen werden als einzelnes und vorzeichenloses Byte übergeben, wenn die Aufzählung weniger als 256 verschiedene Werte enthält, ansonsten im Format *Word*.

- Parameter im Format *Real* werden – obwohl sie sechs Bytes beanspruchen – als Werte übergeben, um die Kompatibilität zur Version 3.0 zu erhalten.
- Parameter im Format *Single* werden als Werte mit vier Bytes übergeben.
- Dasselbe gilt für alle anderen Gleitkommaformate: obwohl *Double*, *Extended* und *Comp* mehr als 4 Bytes Speicherplatz beanspruchen, erfolgt ihre Übergabe als Wert (und damit über den Stack des 8086).

Hinweis: Die Version 4.0 verwendet für *Single..Comp* den Rechenstack des 8087 zur Übergabe und begrenzt damit die Anzahl möglicher Gleitkomma-Parameter bei Aufrufen auf 8. Dieser Mechanismus wird ab Version 5.0 aus Gründen der Kompatibilität zu anderen Programmiersprachen nicht mehr benutzt.

- Zeiger-Typen werden als zwei Werte des Typs *Word* übergeben, wobei Turbo Pascal den Offset-Anteil als letztes auf den Stack legt – dieser Wert steht während der Übergabe also auf der niedrigeren Adresse innerhalb des Stack-Segments.
- String-Typen werden grundsätzlich als Zeiger auf den tatsächlichen Speicherplatz übergeben.
- Mengen-Typen werden als Zeiger auf einen Bereich übergeben, der grundsätzlich 32 Bytes umfaßt (innerhalb dieses Bereichs sind auch Teilmengen so gespeichert, als würden sie Platz für 256 Elemente bieten).
- Arrays und Records werden im Format ihrer Speicherung auf dem Stack übergeben, wenn ihre Gesamtgröße zwischen einem und vier Bytes liegt. Ansonsten erfolgen auch Wertübergaben durch Zeiger.

Funktions- ergebnisse

Ordinale Ergebnisse (Integer-Typen, *Char*, *Boolean* und Aufzählungstypen) werden über die Register des Prozessors zurückgeliefert: einzelne Bytes in AL, Worte in AX und Doppelworte im Registerpaar DX:AX, wobei DX das höherwertige Wort enthält.

- Ergebnisse im Format *Real* werden durch drei Prozessor-Register zurückgeliefert: das höchstwertige Wort in DX, das mittlere in BX und das niederwertige in AX (DX:BX:AX).
- Ergebnisse in den anderen Real-Formaten (*Single*, *Double*, *Extended* und *Comp*) werden nicht über die Register des Prozessors, sondern über das jeweils oberste Register des 8087 (ST(0), »TOS«) zurückgeliefert.
- Zeiger-Typen werden (wie *LongInt*) über das Registerpaar DX:AX zurückgeliefert: DX enthält das Segment, AX den Offset-Anteil.
- Für String-Typen muß die aufrufende Funktion vor den eigentlichen Parametern einen Zeiger auf einen temporären Speicherbereich auf den Stack legen.

Die Funktion liefert ihr Ergebnis in diesem Bereich zurück und darf den Zeiger (im Gegensatz zu allen anderen Parametern) nicht vom Stack entfernen.

Andere Funktionsergebnisse sind nicht erlaubt.

Aufrufe als near und far

Die Prozessoren der 8086-Familie unterscheiden zwischen zwei Arten des Aufrufs bzw. Rücksprungs, die man als *near* und *far* bezeichnet. Aufrufe und Rücksprünge mit *near* laufen innerhalb eines einzigen Code-Segments ab, bei der Codierung als *far* kann mit verschiedenen Segmenten gearbeitet werden.

Ein als *near* codierter Aufruf-Befehl (CALL) legt eine Rücksprungadresse mit 16 Bit auf den Stack, die nur den Offset-Anteil innerhalb des momentanen Code-Segments enthält. Ein als *far* codierter Aufruf (CALL FAR) legt zusätzlich den momentanen Inhalt des Codesegment-Registers (CS) auf dem Stack ab und setzt dieses Register danach neu, benötigt also 32 Bit Speicherplatz. Die entsprechenden Rücksprung-Befehle (RET und RET FAR) lesen die auf dem Stack abgelegten Werte und setzen damit die Prozessor-Register (IP bzw. CS:IP).

Abhängig von der Deklaration der Routine wählt Turbo Pascal automatisch die richtige Art des Aufrufs: wenn die Routine im Interface-Teil einer Unit deklariert ist, wird sie als *far* codiert (und aufgerufen). Eine Deklaration innerhalb des Implementations-Teils oder des Hauptprogramms hat dagegen die Codierung und den Aufruf als *near* zur Folge – derartige Prozeduren und Funktionen können nur innerhalb der Unit bzw. des Programms aufgerufen werden, in dem sie selbst deklariert sind.

Es kann natürlich Gründe geben, warum man eine Routine abweichend von diesen generellen Regeln als *near* oder *far* deklariert – wenn der Aufruf beispielsweise über einen Zeiger geschieht, muß die Codierung grundsätzlich als *far* erfolgen (also bei Exit-Prozeduren, Textdatei-Treibern, den meisten Fehlerbehandlungsroutinen sowie bei Overlays). Nach dem Befehl `{F+}` oder mit Hilfe der **far**-Direktive erzeugt der Compiler immer »weite« Rücksprünge (und merkt sich auch, daß die entsprechenden Routinen als *far* aufgerufen werden müssen). Mit `{F-}` – der Standardvorgabe von Turbo Pascal – wird die zuvor beschriebene Automatik wieder aktiviert. Dies ist bei Verwendung der **far**-Direktive nicht notwendig.

Hinweis: Mit `{F-}` läßt sich eine »bedingungslose« Codierung als *near* nicht erzwingen – im Interface-Teil einer Unit ist dieser Befehl also wirkungslos.

Eingeschachtelte Routinen

Eine Prozedur oder Funktion wird als »eingeschachtelt« bezeichnet, wenn sie innerhalb einer anderen Prozedur oder Funktion deklariert ist (also lokal dazu). In diesem Fall ist sichergestellt, daß sich aufrufende und aufgerufene Routine im selben Code-Segment befinden – deshalb nimmt der Compiler hier eine weitere Optimierung vor und codiert diese Aufrufe auch dann als *near*, wenn der Modus `{F+}` gesetzt ist. Zu dieser Regel gibt es eine Ausnahme: Routinen,

die sich in Overlay-Units befinden, werden grundsätzlich als *far* codiert – unabhängig von ihrer Verschachtelungstiefe.

Beim Aufruf einer lokal deklarierten Prozedur oder Funktion erzeugt der Compiler direkt vor dem CALL den Befehl PUSH BP, übergibt also das BP-Register der aufrufenden Routine als zusätzlichen Parameter. Nachdem die aufgerufene Routine ihren eigenen Stack (mit PUSH BP/MOV BP,SP) eingerichtet hat, kann auf das BP-Register der *aufrufenden* Routine über [BP+4] (*near*) bzw. [BP+6] (*far*) und damit auf die lokalen Variablen dieser übergeordneten Routine zugegriffen werden. (Wenn die aufrufende Routine ihrerseits eingeschachtelt ist, wird die Verbindung zur nächsthöheren Ebene wiederum über [BP+4] bzw. [BP+6] hergestellt usw). Ein Beispiel dazu:

```

procedure A; near;
var
  IntA: Integer;

procedure B; far;
var
  IntB: Integer;

procedure C; near;
var
  IntC: Integer;
begin
  inLine($8B/$46/      / { MOV AX, [BP+IntC]   ; AX = IntC }
        $8B/$5E/$04/    { MOV BX, [BP+4]       ; BX = B's Stack }
        $36/$8B/$47/    / { MOV AX, SS:[BX+IntB]; AX = IntB }
        $8B/$5E/$04/    { MOV BX, [BP+4]       ; BX = B's Stack }
        $36/$8B/$5F/$04/ { MOV BX, SS:[BX+6]    ; BX = A's Stack }
        $36/$8B/$47/    A);{ MOV AX, SS:[BX+IntA]; AX = IntA }

  end {C};

begin {B}
end; {B}
begin {A}
end; {A}

```

Hinweis: Eingeschachtelte Prozeduren und Funktionen können nicht als **external** deklariert werden; die Zuordnung prozeduraler Variablen bzw. ihre Übergabe als prozedurale Parameter ist nicht erlaubt.

Start- und Endcode von Prozeduren und Funktionen

Jede Prozedur oder Funktion von Turbo Pascal beginnt und endet mit einigen standardisierten Befehlen:

```

PUSH BP          ; BP auf den Stack
MOV BP,SP        ; Stand des Stackzeigers --> BP-Register
SUB SP, LocalSize ; Platz für lokale Variablen schaffen

```

```

....
....   { Prozedur- oder Funktionsblock: }
....   { Übergebene Parameter und lokale Variablen }
....   { werden via BP adressiert }
....
....   { Für Funktionen: MOV AX, Ergebnis[BP] }
....
MOV  SP,BP      ; Stackzeiger zurück -> Freigabe lokaler Variablen
POP  BP         ; BP wiederherstellen
RET  ParamSize  ; Parameter vom Stack und Rücksprung

```

LocalSize steht hier für die Menge an Speicherplatz, die für lokale Variablen benötigt wird – wenn die Routine ohne lokale Variablen arbeitet, dann fehlt der SUB-Befehl ganz. Das Register BP hält den Stand des Stackzeigers (SP) zu Beginn der Routine fest und bleibt während der gesamten Ausführung der Prozedur/Funktion bzw. wird vor Veränderungen zwischengespeichert. Wenn die Prozedur als *near* codiert ist, beginnt der als letztes deklarierte Parameter mit der Adresse *BP+04*, bei einer Codierung als *far* mit *BP+06*.

Vor dem Rücksprung wird der ursprüngliche Inhalt des Stackzeigers wieder hergestellt (und damit der durch lokale Variablen belegte Speicherplatz wieder freigegeben). *ParamSize* steht für die Anzahl der Bytes, die durch übergebene Parameter belegt wurden – sie werden durch den Rücksprungbefehl entfernt, der je nach Art des Aufrufs entweder als RET oder als RET FAR codiert ist.

Verfügbare Prozessor-Register

Die Register BP, SP, SS und DS müssen während der Ausführung einer Funktion oder Prozedur konstant gehalten bzw. vor Veränderungen zwischengespeichert werden. Alle anderen Register sind frei verwendbar.

Exit- Prozeduren

Durch die Installation einer Exit-Prozedur können Sie festlegen, wie sich ein Programm nach seinem »offiziellen« Ende verhält. Sinnvoll ist so etwas speziell dann, wenn Sie dafür sorgen müssen, daß bestimmte Aktionen unabhängig von der Art des Programmendes auf jeden Fall ausgeführt werden (wie beispielsweise das Zurücksetzen von Interrupt-Vektoren oder das Schließen von Dateien).

Turbo Pascal deklariert eine globale Zeigervariable namens *ExitProc*, über die bei jeder Art von Programmende ein Aufruf stattfindet – egal, ob das Programm normal, durch *Halt* oder durch einen Laufzeitfehler beendet wird.

Über eine explizite Zuweisung an *ExitProc* können Sie eine eigene Exit-Prozedur festlegen, die zwei Voraussetzungen erfüllen muß: zum einen darf sie keine

Parameter erwarten, zum anderen muß sie mit dem Befehl `{$F+}` kompiliert worden sein (also als *far*).

Die Zuweisung der Startadresse dieser Prozedur an *ExitProc* besteht aus zwei Schritten: dem Speichern des vorherigen Wertes und der eigentlichen Zuweisung selbst. Tatsächlich sieht die Laufzeitbibliothek von Turbo Pascal nämlich nicht nur eine Prozedur zum Beenden eines Programms vor, sondern eine *Prozedurkette* – speziell dann, wenn Units mit aufgenommen worden sind, deren Initialisierungs-Teil diverse Systemparameter verändert hat (wie beispielsweise *Crt*). Diese Units verlassen sich darauf, daß ihre eigene Exit-Prozedur beim Beenden des Programms aufgerufen wird, um Veränderungen (wie Interrupt-Vektoren, direkte Speicherzugriffe etc.) wieder rückgängig zu machen.

Aus diesem Grund werden auch an den Code einer eigenen Exit-Prozedur noch einige Anforderungen gestellt, die das folgende Programmfragment illustriert:

```

program ExitTest;
var
    ExitSave: Pointer; { für die originale Exit-Prozedur }

procedure MyExit; far;
begin
    ExitProc := ExitSave; { ExitProc wieder aufs Original - sollte als
                          * erster Schritt* geschehen }
    ...
    ...
end;

begin { Hauptprogramm }
    ExitSave := ExitProc; { Original-Zeiger speichern }
    ExitProc := @MyExit; { und die eigene Routine installieren }
    ...
    ...
end.

```

Zu Beginn des Programms wird also der ursprüngliche Inhalt von *ExitProc* in einer globalen Variablen gespeichert, darauf folgt die Installation der eigenen Exit-Prozedur. Innerhalb von *MyExit* muß dieser Original-Inhalt auf jeden Fall wieder gesetzt werden, bevor das Programm dann tatsächlich endet.

Die Routine der Laufzeitbibliothek, die für das Beenden von Programmen zuständig ist, führt solange Aufrufe über *ExitProc* aus, bis dieser Zeiger den Wert **nil** bekommt. Damit keine Endlosschleifen entstehen können, wird *ExitProc* vor dem Aufruf jeder Exit-Prozedur auf **nil** gesetzt. Wenn Ihre Prozedur also »vergißt«, den zu Beginn des Programms gespeicherten Wert wiederherzustellen, wird die Kette dadurch abgebrochen.

Ein weiteres Detail: Die Zuweisung an *ExitProc* sollte direkt zu Beginn der Exit-Prozedur stattfinden. Ansonsten können Endlosschleifen entstehen: ein Abbruch der Exit-Prozedur durch einen Laufzeitfehler wird als Programmende gewertet, dadurch die Exit-Prozedur aufgerufen, die wiederum einen Laufzeitfehler erzeugt usw.

Für die gezielte Behandlung von Fehlern gibt es eine weitere Möglichkeit:

eine Exit-Prozedur kann den »Grund« des Programmendes über die Variablen *ExitCode* und *ErrorAddr* ermitteln. Bei einem normalen Programmende hat *ExitCode* den Wert 0, bei einem Abbruch durch *Halt* enthält diese Variable das Argument von *Halt* (*ErrorAddr* zeigt in beiden Fällen auf **nil**). Beim Abbruch durch einen Laufzeitfehler enthält *ExitCode* die entsprechende Fehlernummer, *ErrorAddr* zeigt auf den Befehl, der den Fehler verursacht hat.

Die als allerletztes ausgeführte Exit-Prozedur (die durch die Laufzeitbibliothek selbst installiert wird) schließt die Dateien *Input* und *Output*, setzt die durch *System* veränderten Interrupt-Vektoren wieder zurück und prüft *ErrorAddr*: falls hier nicht der Wert **nil** gespeichert ist, wird eine entsprechende Meldung ausgegeben.

Wenn Sie eigene Meldungen für Laufzeitfehler einsetzen wollen, können Sie eine Exit-Prozedur schreiben, die *ErrorAddr* überprüft, die entsprechende Meldung ausgibt und danach *ErrorAddr* auf **nil** setzt, um weitere Fehlermeldungen zu verhindern.

Und schließlich: Nachdem alle Exit-Prozeduren ausgeführt worden sind, folgt der Rücksprung zu DOS, wobei der Wert von *ExitCode* als Ergebnis des Programms übergeben wird.

Bearbeitung von Interrupts

Der durch den Compiler erzeugte Code und die Routinen der Laufzeitbibliothek sind interruptfähig, d.h. können beliebig durch Interrupts unterbrochen werden. Der größte Teil der Laufzeitbibliothek ist darüber hinaus reentrant – es ist also möglich, Interrupt-Prozeduren in Turbo Pascal zu schreiben.

Interrupt-Prozeduren

werden in Turbo Pascal durch das reservierte Wort **interrupt** deklariert. Der Kopf einer **interrupt**-Prozedur ist standardisiert:

```
procedure <Name> (Flags, CS, IP, AX, BX, CX, DX, SI, DI, DS, ES, BP: Word);  
interrupt;
```

Alle Register des Prozessors werden hier als (Pseudo-)var-Parameter übergeben und können innerhalb der Prozedur verändert werden. Es ist möglich, die Parameterliste zu verkürzen – aber nur in einer ganz bestimmten Form:

```
procedure <Name> (      CS, IP, AX, BX, CX, DX, SI, DI, DS, ES, BP: Word);  
interrupt;  
procedure <Name> (      AX, BX, CX, DX, SI, DI, DS, ES, BP: Word);  
interrupt;  
.....  
.....
```

```

procedure Name> (                                ES,BP: Word);
interrupt;
procedure Name> (                                BP: Word);
interrupt;

```

Anders gesagt: die »vorderen« Parameter dürfen ausgelassen werden, solange die Reihenfolge der restlichen Angaben eingehalten wird. Überzählige Parameter haben meistens einen Systemabsturz zur Laufzeit des Programms zur Folge. Ähnlich katastrophal endet der Versuch, ein oder mehrere »Löcher« in der Liste zu erzeugen, also

```

procedure Name> (Flags, CS,          CX,DX,SI,          ES,BP: Word);
interrupt      { *** FALSCH !!! *** }

```

Vorsicht: Derartige Fehler werden vom Compiler nicht entdeckt.

Unabhängig von der Deklaration der Parameter beginnt der Compiler jede als **interrupt** deklarierte Prozedur mit der Speicherung aller Register auf dem Stack und dem Setzen des DS-Registers auf das Datensegment des Programms:

```

PUSH  AX
PUSH  BX
PUSH  CX
PUSH  DX
PUSH  SI
PUSH  DI
PUSH  DS
PUSH  ES
PUSH  BP
MOV   BP,SP
SUB   SP,LocalSize    ; Platz für lokale Variablen
MOV   AX,SEG DATA    ; Datensegment-Startadresse
MOV   DS,AX           ; Turbo Pascal-Datensegment -> DS-Register

```

Einen CLI-Befehl, der weitere Interrupts sperrt, werden Sie hier nicht finden – falls er notwendig ist, müssen Sie ihn (via **inline** oder **asm**) selbst codieren. Der Code, mit dem der Compiler eine **interrupt**-Prozedur beendet, ist ebenfalls standardisiert:

```

MOV   SP,BP          ; Freigabe lokaler Variablen
POP   BP             ; und danach alle Register wieder vom Stack
POP   ES
POP   DS
POP   DI
POP   SI
POP   DX
POP   CX
POP   BX
POP   AX
IRET                ; Rücksprung

```

Zugriffe auf die Parameter einer **interrupt**-Funktion sprechen die auf dem

Stack gespeicherten Werte an: im Prinzip handelt es sich also um eine **var**-Deklaration (auch wenn sie formal nicht als solche ausgewiesen ist). Ein Befehl wie

```
AX := $4421;
```

innerhalb der **interrupt**-Prozedur sorgt also dafür, daß das AX-Register vor dem Rücksprung auf den Wert \$4421 gesetzt wird. Auf diese Weise kann auch eine Interrupt-Routine Ergebnisse zurückliefern.

interrupt-Prozeduren, die durch Hardware-Ereignisse (wie z.B. den »Clock Tick« der eingebauten Uhr) ausgelöst werden, müssen ohne die Bibliotheks-routinen für Ein- und Ausgaben, die dynamische Speicherverwaltung und Aufrufe von Overlay-Routinen auskommen, weil der entsprechende Code nicht reentrant ist. Dasselbe gilt für DOS-Funktionen: Unterbrechungen durch Interrupts sind jederzeit möglich – wenn aber die dadurch aktivierte Routine DOS erneut aufruft, kommt das System gründlich durcheinander.

Hinweis: **interrupt**-Prozeduren verwenden den Stack des unterbrochenen Programms, das entsprechend Platz zur Verfügung stellen muß.

Ein Beispiel für eine **interrupt**-Prozedur:

```
procedure CtrlBreakHandler; interrupt;
begin
  CtrlBreakPressed:= True;
end;
```

Gerade bei **interrupt**-Prozeduren dieser Art kann es sinnvoll sein, sie durch »interrupt-fähige« **assembler**-Prozeduren zu ersetzen, die nur solche Register retten, welche tatsächlich verändert werden. Das folgende Beispiel spart im Vergleich zur vorigen 16 von 31 Bytes Code:

```
procedure CtrlBreakHandler; assembler;
asm
  PUSH  AX
  PUSH  DS
  MOV   AX, SEG @Data
  MOV   DS, AX
  MOV   CtrlBreakPressed, 1
  POP   DS
  POP   AX
  IRET
end;
```

Details zu **assembler**-Prozeduren finden Sie im nächsten Kapitel »Der integrierte Assembler und Einbinden von Assembler-Code«.

Textdatei- Gerätetreiber

Wie bereits in Kapitel 12 »Standard-Units« erwähnt, erlaubt Turbo Pascal die Definition eigener Gerätetreiber für Textdateien. Ein solcher »Treiber« besteht aus einem Satz von vier Funktionen, die als Schnittstelle zwischen Turbo Pascal und einem beliebigen Gerät dienen. Die vier Funktionen übernehmen die Ausführung der Operationen *Open*, *InOut* (Ein-/Ausgabe), *Flush* (erzwungenes Schreiben eines Pufferinhalts) und *Close*. Alle vier Funktionsköpfe haben dasselbe Format:

```
function <Name>(var F: TextRec): Integer;
```

Der Datentyp *TextRec* ist in der Unit *Dos* definiert und wurde bereits im Abschnitt »Datei-Typen« besprochen. Er kann (allerdings mit äußerster Sorgfalt!) redefiniert werden, wie wir es in den folgenden Beispielen demonstrieren.

Die Namen der vier Funktionen sind beliebig – allen gemeinsam ist, daß sie mit dem Befehl *{F+}* oder mit Hilfe der *far*-Direktive compiliert werden, einen **var**-Parameter des Typs *TextRec* (bzw. einer Modifikation davon) erwarten und einen Integer zurückliefern müssen.

Um die Verbindung zwischen dem Treiberprogramm und dem dadurch angesprochenen Gerät herzustellen, müssen Sie eine eigene *Assign*-Prozedur schreiben. Der Name dieser Routine ist ebenfalls beliebig – er sollte eine logische Verbindung zu seiner Funktion darstellen (wie beispielsweise *AssignCrt* in der Unit *Crt*). Eine eigene Version von *Assign* muß die folgenden Schritte ausführen:

- Setzen der vier Zeiger in der Datei-Variablen mit den Adressen der vier Funktionen (*Open*, *InOut*, *Flush* und *Close*).
- Setzen des Feldes *Mode* mit der »magischen« Konstanten *fmClosed*.
- Löschen des Eintrags im Feld *Name*, Speicherung der Größe und der Adresse des (vordefinierten) Puffers in den Feldern *BufSize* bzw. *BufPtr*.

Wenn wir davon ausgehen, daß die vier Funktionen die Namen *DevOpen*, *DevInOut*, *DevFlush* und *DevClose* haben, dann könnte die Prozedur *DevAssign* so aussehen:

```
procedure DevAssign(var F: Text);
begin
  with TextRec(F) do { Typ-Umwandlung in TextRec! }
    begin
      Mode      := fmClosed;
      BufSize   := SizeOf(Buffer);
      BufPtr    := @Buffer;
      OpenFunc  := @DevOpen;
      InOutFunc := @DevInOut;
      FlushFunc := @DevFlush;
      CloseFunc := @DevClose;
      Name[0]   := #0;
```

```
end;  
end;
```

Die vier Funktionen des Gerätetreibers können das Feld *UserData* der Datei-Variablen zur Speicherung eigener Informationen benutzen – dieses Feld wird von Turbo Pascal selbst nicht verwendet.

Die Open-Funktion

wird durch die Standard-Prozeduren *Reset*, *Rewrite* und *Append* aufgerufen. Sie öffnet die mit der Datei-Variablen verbundene Datei bzw. initialisiert ein angeschlossenes Gerät. Vor ihrer Ausführung wird das Feld *Mode* auf einen der drei Werte *fmInput*, *fmOutput* oder *fmInOut* gesetzt und zeigt so an, durch welche Prozedur der Aufruf erfolgt ist.

Open muß grundsätzlich vor allen anderen Funktionen des Gerätetreibers aufgerufen werden. Die Standardprozedur *Assign* nützt diese Tatsache aus: sie initialisiert nur das Feld *OpenFunc* der Datei-Variablen und überläßt es *Open*, die restlichen Funktionszeiger abhängig von *Mode* zu setzen. *InOut*, *Flush* und *Close* müssen daher nicht erst selbst bestimmen, in welcher Weise sie arbeiten sollen.

Die InOut-Funktion

wird durch *Read*, *Readln*, *Write*, *Writeln*, *Page*, *Eof*, *Eoln*, *SeekEof*, *SeekEoln* und *Close* aufgerufen und übernimmt den gesamten Datentransfer zwischen dem Programm und dem angeschlossenen Gerät.

Die Arbeitsweise von *InOut* wird durch *Mode* bestimmt: wenn *Mode* den Wert *fmInput* hat, dann liest *InOut* maximal *BufSize* Zeichen in den Puffer der Datei-Variablen (*^BufPtr*) und liefert die Anzahl der gelesenen Zeichen indirekt über *BufEnd* zurück. Nach jedem neuen Setzen des Pufferinhalts mit gelesenen Daten muß sie *BufPos* auf 0 setzen; wenn sie *BufEnd* als Ergebnis einer Eingabeanforderung auf 0 setzt, dann liefern nachfolgende Aufrufe von *Eof* den Wert *True* zurück.

Wenn *Mode* auf den Wert *fmOutput* gesetzt ist, dann schreibt die Funktion *InOut* bei jedem Aufruf die durch *BufPos* angegebene Anzahl von Zeichen zu dem Gerät. Die Daten werden via *^BufPtr* gelesen; nach dem Beenden der Aktion wird *BufPos* auf 0 zurückgesetzt.

Die Flush-Funktion

wird automatisch nach jedem *Read*, *Readln*, *Write*, *Writeln* oder direkt (durch die Standard-Prozedur *Flush*) aufgerufen. Ihre Arbeitsweise ist ebenfalls von *Mode* abhängig:

Wenn *Mode* den Wert *fmInput* hat, dann setzt *Flush* die Felder *BufPos* und *BufEnd* auf 0, ignoriert also sämtliche Zeichen, die sich eventuell noch im Puffer befinden. (Diese Möglichkeit ist allerdings selten praktisch verwendbar.)

Wenn *Mode* den Wert *fmOutput* hat, dann schreibt *Flush* (auf dieselbe Weise wie *InOut*) den Inhalt des Puffers und stellt so sicher, daß sämtliche vom Programm ausgegebenen Zeichen ohne Verzögerung zum angeschlossenen Gerät gesendet werden.

Die durch *Flush* durchgeführten Operationen sind optional; falls diese Funktion lediglich als »Dummy« definiert ist, werden Ausgaben erst dann physikalisch geschrieben, wenn der Puffer voll ist oder die Datei geschlossen wird.

Die Close-Funktion

wird durch die Standard-Prozedur *Close* aufgerufen und schließt die Datei. (*Reset*, *Rewrite* und *Append* rufen *Close* ebenfalls auf, wenn die Datei zum entsprechenden Zeitpunkt bereits offen ist.)

Wenn *Mode* den Wert *fmOutput* hat, ruft das Dateisystem von Turbo Pascal vor *Close* zuerst einmal *InOut* auf, um eventuell noch im Puffer befindliche Zeichen zu schreiben.

Direkte Zugriffe auf I/O-Adressen

Die Arrays *Port* und *PortW* *Port* und *PortW* sind als **array of Byte** bzw. **array of Word** deklariert und erlauben direkte Zugriffe auf die Ein-/Ausgabeadressen des Systems. Beide Arrays sind eindimensional, über den Index (*Typ Word*) wird die entsprechende I/O-Adresse festgelegt. Zuordnungen von Werten zu Komponenten der Arrays *Port* und *PortW* haben eine Ausgabe (OUT-Befehl) zur entsprechenden I/O-Adresse zur Folge, Bezüge auf Komponenten dieser Arrays innerhalb eines Ausdrucks führen eine Leseaktion (IN-Befehl) durch. Einige Beispiele:

```
Port[$20] := $FF;           { schreibt zur I/O-Adresse $20 }
Port[Base] := Port[Base] xor Maske;
while Port[$B2] and $80 do ... { war ten }
Index := $15
PortW[Index] := 0;          { schreibt zur I/O-Adresse Index }
```

Port und *PortW* können nur gelesen und beschrieben werden – die Verwendung einzelner Komponenten als **var**-Parameter oder eine Bezugnahme auf das gesamte Array (d.h. ohne Index) ist nicht erlaubt.

Automatische Optimierung des Programmcodes

Turbo Pascal optimiert den erzeugten Code in mehrfacher Hinsicht automatisch, wobei sich die angewendeten Verfahren in verschiedene Klassen einteilen lassen. Die folgenden Abschnitte beschreiben die Einzelheiten.

Vorausberechnung von Konstanten

Konstanten-Ausdrücke, die sich aus ordinalen Werten zusammensetzen, werden von Turbo Pascal durch den Compiler berechnet: eine Zuweisung wie

```
X := 3 + 4 * 2;
```

erzeugt deshalb exakt denselben Code wie

```
X := 11;
```

Dasselbe gilt nicht nur für (ordinale) Argumente der Funktionen *Abs*, *Sqr*, *Succ*, *Pred*, *Odd*, *Lo*, *Hi* und *Swap*, bei denen der Compiler gleich das Ergebnis einsetzt, sondern auch für das Verbinden von Strings:

```
XStr:= 'Ein ' + 'Text';
```

erzeugt denselben Code wie

```
XStr := 'Ein Text';
```

Indizierungen von Arrays, die über Konstanten definiert sind, werden vom Compiler als absolute Adressen berechnet: ein Ausdruck wie *Index*[5,5] erzeugt also denselben Code wie der Zugriff auf eine einfache Variable.

Gemeinsame Speicherung mehrfach verwendeter Konstanten

String-Konstanten, die mehrfach innerhalb eines Programms angegeben sind, werden nur einmal gespeichert. Ein Beispiel:

```
if (ch = 'E') then begin
  writeln('Programm beendet.');
```

```
  writeln('Auf Wiedersehen!');
```

```
end
```

```
else if (ch = 'X') then begin
```

```
  writeln('Programm abgebrochen...');
```

```
  writeln('Auf Wiedersehen!');
```

```
end;
```

Hier reserviert der Compiler für die Zeichenfolge *Auf Wiedersehen!* nur einmal Speicherplatz – der Aufruf von *Writeln* in der vorletzten Zeile enthält lediglich

einen weiteren Verweis auf die Adresse, die bereits in der dritten Programmzeile verwendet wird. Der Compiler vergleicht allerdings nur vollständige Strings miteinander – die Zeichenfolge *Program* wird im gezeigten Beispiel also zweimal gespeichert.

Auswertung zusammengesetzter boolescher Ausdrücke

Turbo Pascal unterstützt zwei Arten der Code-Erzeugung für die Auswertung zusammengesetzter Ausdrücke mit den Operatoren **and** und **or**: vollständige Berechnung und das Kurzschlußverfahren:

■ Bei einer vollständigen Berechnung werden sämtliche Teile des Ausdrucks berechnet – selbst dann, wenn das Ergebnis bereits feststeht, also beispielsweise der erste Teil einer **and**-Operation den Wert *False* ergibt. Diese Art der Auswertung ist z. B. erforderlich, wenn Teile des Ausdrucks aus Funktionsaufrufen bestehen, über die weitere Operationen ausgeführt und/oder Werte verändert werden (Seiteneffekt).

■ Das Kurzschlußverfahren garantiert eine strikte Auswertung von links nach rechts und bricht ab, sobald das Ergebnis des Gesamtausdrucks feststeht – beispielsweise dann, wenn der erste Teil einer **or**-Operation den Wert *True* ergibt. Diese Art der Auswertung erhöht nicht nur die Arbeitsgeschwindigkeit des Programms und erzeugt normalerweise kompakteren Code, sondern ermöglicht auch Konstrukte, die bei vollständiger Berechnung nicht zulässig wären. Zwei Beispiele dazu:

```

while (K <> nil) and (K <> ' ') do Inc(I);
while (K <> nil) or (K <> 5) do P := P^.Next;

```

Hier wird in beiden Fällen beim Kurzschlußverfahren die zweite Prüfung nicht durchgeführt, wenn die erste den Wert *False* ergibt. Bei einer vollständigen Berechnung ergäben sich Zugriffe auf nicht definierte Adressen als Teil der Prüfung selbst (vgl. Kapitel 2 »Compiler-Befehle« des *Referenzhandbuchs*).

Welche Art von Code der Compiler zur Auswertung zusammengesetzter Bedingungen generiert, wird über den Schalter *\$B* kontrolliert, der standardmäßig auf *\$B-* (Kurzschlußverfahren) gesetzt ist. Nur wenn dieser Schalter (mit *\$B+* oder via *Options/Compiler*) aktiviert wird, berechnet das Programm zusammengesetzte logische Ausdrücke vollständig.

Reihenfolge der Berechnung

Wie durch den ANSI-Standard festgelegt, können Teile zusammengesetzter Ausdrücke unabhängig von der Reihenfolge ihrer Angabe berechnet werden. Wenn wir davon ausgehen, daß *F* und *G* Funktionen mit dem Ergebnistyp *Integer* sind, dann codiert der Compiler in einer Zuweisung wie

```
I := F(J) div G(J);
```

den Aufruf der Funktion *G* zuerst, weil dadurch die abschließende Division einfacher auszuführen ist. Aus diesem Grund schreibt die Sprachdefinition von Pascal vor, daß derartige Ausdrücke nicht mit Seiteneffekten arbeiten dürfen: wenn der Wert von *J* im oben gezeigten Beispiel durch den Aufruf einer der beiden Funktion verändert wird, hätte *I* sonst ein Ergebnis, das von der Arbeitsweise des Compilers abhängig wäre.

Falls es notwendig sein sollte, mit Seiteneffekten zu arbeiten, müssen Sie eine temporäre Variable und voneinander getrennte Funktionsaufrufe verwenden, also:

```
Temp := F(J); I := Temp div G(J);
```

Hinweis: Das Kurzschlußverfahren bei der Bewertung boolescher Ausdrücke stellt eine Ausnahme von dieser Regel dar – nach der Aktivierung mit {\$B-} werden über **and** und **or** verbundene Operanden *immer* von links nach rechts ausgewertet.

Überprüfung von Wertebereichen

Indizierungen, Konstanten-Zuweisungen an Variablen sowie Prozedur- und Funktionsaufrufe mit Konstanten als Parameter werden während der Compilierung daraufhin geprüft, ob sie innerhalb der zulässigen Wertebereiche liegen – dadurch wird der entsprechende Code zur Laufzeit des Programms eingespart. Das folgende Programm enthält drei Fehler, die alle bereits vom Compiler entdeckt werden:

```
var
  X: Byte;
  AX: array[1..10] of Byte;
procedure Dummy (S: ShortInt);
begin
  ...
end;

begin
  AX[12] := 100; { unzulässige Indizierung }
  X := 999;      { Zuweisung außerhalb des Wertebereichs }
  Dummy(200);    { Parameter außerhalb des Wertebereichs }
end.
```

Schiebebefehle anstelle von Multiplikationen

Integer-Multiplikationen der Form $X * C$ werden über den Prozessorbefehl SHL codiert, wenn *C* eine Konstante mit dem Wert einer Zweierpotenz ist.

In ähnlicher Weise geschieht die Indizierung von Arrays: wenn die Größe eines Array-Elements als Zweierpotenz darstellbar ist (2, 4, 8 ... 256 Bytes), dann verwendet Turbo Pascal zur Berechnung der absoluten Adresse den Befehl SHL anstelle einer Multiplikation.

Wortweise Ausrichtung von Daten

Mit der Standardvorgabe `{$A+}` sorgt der Compiler dafür, daß sämtliche Variablen außer *Char*, *Byte* und *ShortInt* auf geradzahlgigen Adressen im Speicher beginnen, und macht so Zugriffe des Prozessors 8086 und seiner Nachfolger schneller. Auf Maschinen mit einem 8088, die nur über einen Datenbus mit 8 Bit Breite verfügen, hat diese Optimierung keine Wirkung (vgl. Menüpunkt *O/C/ Align data* in Kapitel 7 »Die IDE-Referenz« des *Benutzerhandbuches*).

Entfernung »toten« Codes

Befehle und Anweisungen, die aus logischen Gründen nie ausgeführt werden können, entfernt der Compiler selbständig. Zwei Beispiele:

```
const
  MonoCRT = False;
...
if MonoCRT then ...
  while false do ...
```

Hier wird weder für die **if**-Bedingung noch für die **while**-Schleife Code erzeugt.

Intelligentes Linken

Der in Turbo Pascal eingebaute Linker nimmt nur die Funktionen und Prozeduren in die .EXE-Datei auf, die auch tatsächlich verwendet werden – Routinen, für die das Programm keine Aufrufe enthält, werden automatisch entfernt. (Aus diesem Grund erzeugt auch die Aufnahme einer so komplexen Unit wie *Graph* nur die tatsächlich benötigte Menge an Code).

Analoges gilt für Datendeklarationen, auf die das Programm nicht zugreift. Das »Aussortieren« unnötigen Codes arbeitet auf der Basis einzelner Routinen; das Entfernen von Daten dagegen auf der Basis einzelner Deklarations-Abschnitte. Ein Beispiel zur Verdeutlichung:

```

program SmartLink;

const
  H: array[0..15] of Char = '0123456789ABCDEF';

var
  I,J: Integer;
  X,Y: Real;

var
  S: string[79];

var
  A: array[1..10000] of Integer;

procedure P1;
begin
  A[1] := 1;
end;

procedure P2;
begin
  I := 1;
end;

procedure P3;
begin
  S := 'Turbo Pascal';
  P2;
end;

begin
  P3;
end.

```

Das Hauptprogramm ruft hier die Prozedur *P3* auf, die ihrerseits einen Aufruf von *P2* enthält – folglich werden diese beiden Routinen in die .EXE-Datei aufgenommen. *P2* greift mit der Zuweisung an *I* auf die erste **var**-Sektion des Programms zu, *P3* mit der Zuweisung an *S* auf die zweite – die .EXE-Datei enthält damit die Variablen *I*, *J*, *X*, *Y* und *S*.

Die Prozedur *P1* wird an keiner Stelle verwendet und deshalb vom Linker aus der .EXE-Datei hinausgeworfen; dasselbe gilt für die Variable *A* (auf die nur innerhalb der entfernten Routine *P1* zugegriffen wird) und die typisierte Konstante *H*, auf die überhaupt kein Bezug des Programms existiert.

Dieses »intelligente« Verhalten des Linkers macht sich vor allem bei der Aufnahme von Units bemerkbar, die mehr oder weniger als Bibliotheken aufgebaut sind: Die Unit *Dos* deklariert beispielsweise eine recht hohe Zahl von Prozeduren, Funktionen und Datenstrukturen, die praktisch nie in einem Programm auf einmal verwendet werden. Jedes Programm, das *Dos* verwendet, bekommt exakt die Dinge zugewiesen, die tatsächlich notwendig sind – der restliche Code wird zusammen mit überflüssigen Datenstrukturen entfernt.

Hinweis: Bei der Compilierung in den Speicher arbeitet der Linker ohne

»Intelligenz« (und erhält damit den exakten Bezug zwischen Quelltext und Code). Deshalb ist die .EXE-Datei in den meisten Fällen erheblich kleiner, als *Compile/Get info* nach einer Übersetzung mit *Destination..Memory* anzeigt.

Optimierungen durch den Programmierer

Inc und Dec

Diese Prozeduren sind nicht nur die kürzere und schnellere Lösung, wenn es um die Erhöhung bzw. Erniedrigung einfacher Variablen geht (also dem Ersatz von $X := X + N$ durch $Inc(X,N)$). Speziell bei komplexeren Ausdrücken wie $Inc(A[I,J],N)$ kommt der Vorteil deutlich zum Tragen, daß Variablenadressen während der Operation nur einmal berechnet werden müssen.

Boolesche Ausdrücke anstelle von Mengen

Zur Feststellung einer Mengenzugehörigkeit wie in

```
if Ch in 'J','j','N','n' then ...
```

muß das Programm eine Menge (mit 256 Elementen) auf dem Stack erzeugen, die angegebenen Elemente (J , j , N und n) dazugaddieren, die Ordinalität des Wertes von Ch als Teilmenge umrechnen und schließlich prüfen, ob dieser Wert in der Menge enthalten ist, bevor die gesamte Menge wieder vom Stack entfernt wird. Etwas kürzer gesagt: Tests mit booleschen Operatoren wie

```
if (Ch = 'J') or  
    (Ch = 'j') or  
    (Ch = 'N') or  
    (Ch = 'n') then ...
```

sind zwar von der Übersichtlichkeit des Programms her nicht gerade die beste Lösung, werden aber erheblich schneller ausgeführt. Die Prüfung ganzer Bereiche wie

```
if Ch in ['A'..'Z'] then ...
```

läßt sich durch zwei Vergleiche ersetzen, wobei die Änderung des Programms auch nicht auf Kosten der Übersichtlichkeit geht:

```
if (Ch = 'A') and (Ch <= 'Z') then ...
```

Compiler-Schalter

Der Compiler erzeugt automatisch zusätzlichen Prüfcode, um die folgenden Fehler abzufangen:

- nach *jeder* Ein-/Ausgabeoperation wird der Fehlerstatus geprüft – also auch nach einem *Write* oder *Writeln* zum Bildschirm (Schalter $\{ \$I+/- \}$);
- vor jedem Aufruf einer Prozedur oder Funktion wird geprüft, ob noch genügend Platz auf dem Stack vorhanden ist (Schalter $\{ \$S+/- \}$);
- bei der Veränderung von Integer-Variablen und der Indizierung von Arrays wird geprüft, ob sich der neue Wert bzw. der angegebene Index innerhalb des zulässigen Bereichs bewegt (Schalter $\{ \$R+/- \}$);

Diese Prüfungen können während der Entwicklungs- und Testphase eines Programms lebenswichtig sein. Da sie aber den Code-Umfang erhöhen und die Ausführungsgeschwindigkeit mitunter erheblich herabsetzen, sollten sie in einer fertigen Anwendung abgeschaltet ($\{ \$S- \}$) bzw. nur an den Stellen verwendet werden, wo sie wirklich notwendig sind ($\{ \$I+ \}$ und $\{ \$R+ \}$).

15. Der integrierte Assembler und Einbinden von Assembler-Code

Der integrierte Assembler

Der integrierte Assembler von Turbo Pascal erlaubt Ihnen, 8086/8087- bzw. 80286/80287-Assembler-Anweisungen direkt in den Pascal-Quelltext einzufügen. Künftig ist es somit nicht mehr nötig (obwohl immer noch möglich), per Hand Maschinencode für **inline**-Anweisungen zu erzeugen oder **external**-Prozeduren und -Funktionen in Ihr Pascal-Programm einzubinden.

Die Syntax des integrierten Assemblers entspricht weitgehend der Syntax von Turbo Assembler bzw. Microsofts Makro-Assembler. Sein Befehlssatz umfaßt sämtliche Befehle der 8086/8087- und 80286/80287-Prozessoren sowie die meisten Turbo-Assembler-Operatoren.

Mit Ausnahme der Anweisungen *DB*, *DW* und *DD*, unterstützt der integrierte Assembler keine weiteren Turbo-Assembler-Anweisungen (also z.B. weder *EQU*, *PROC*, *STRUC*, *SEGMENT* noch *MACRO*). Meist haben jedoch Operationen, die bei Turbo Assembler über Anweisungen implementiert sind, eine Entsprechung in Turbo-Pascal-Konstrukten. Die meisten *EQU*-Anweisungen können zum Beispiel durch Konstanten-, Variablen- oder Typdeklarationen gleicher Bedeutung ersetzt werden. Die *PROC*-Anweisung findet ihre Entsprechung in Prozedur- und Funktionsdeklarationen, genauso wie die *STRUC*-Anweisung den **record**-Typen von Turbo Pascal gleicht. Somit kann man sich den integrierten Assembler fast als einen Maschinensprache-Compiler vorstellen, der für alle Deklarationen die Pascal-Syntax benutzt.

Die Anweisung *asm*

Assembler-Blöcke werden durch eine **asm**-Anweisung eingeleitet und mit **end** abgeschlossen. Die Syntax von **asm**-Anweisungen lautet:

```
asm  AsmBefehl < Trennzeichen AsmBefehl >  end

if EnableInts then
    asm
        STI
```

```

    end
  else
    asm
      CLI
    end

  asm
    MOV AX,Links;  XCHG AX,Rechts;  MOV Links,AX;
  end;

  asm
    MOV  AX,0                (* Funktion zum Lesen der Tastatur *)
    INT  16H                 (* BIOS-Aufruf *)
    MOV  CharCode,AL         (* ASCII-Code sichern *)
    MOV  ScanCode,AH        (* Scan-Code sichern *)
  end;

  asm
    PUSH DS                 (* Daten-Segment sichern *)
    LDS  SI,Source          (* Quellzeiger laden *)
    LES  DI,Dest            (* Zielzeiger laden *)
    MOV  CX,Count           (* Blockgröße laden *)
    CLD                    (* Direction-Flag setzen *)
    REP  MOVSB              (* Block kopieren *)
    POP  DS                 (* Daten-Segment restaurieren *)
  end;

```

Wie Sie sehen, ist es durchaus zulässig, mehrere Assembler-Anweisungen in eine einzige Zeile zu schreiben. Beachten Sie jedoch, daß Assembler-Anweisungen in einem solchen Fall durch einen Strichpunkt getrennt sein müssen. Beachten Sie bitte auch, daß ein Strichpunkt in einer Zeile mit einer Assembler-Anweisung, entgegen der Konventionen in Assembler, den Rest der Zeile nicht als Kommentar markiert. Kommentare müssen nach wie vor in der üblichen Pascal-Manier eingeleitet bzw. abgeschlossen werden, also mit { und } bzw. mit (* und *).

Einsatz der Register

In einer **asm**-Anweisung können Sie die Prozessor-Register in der gleichen Weise verwenden, wie Sie das von **external**-Prozeduren und -Funktionen her gewohnt sind. Sie haben lediglich dafür Sorge zu tragen, daß die Register BP, SP, SS und DS beim Austritt aus einem Assembler-Block, die gleichen Inhalte aufweisen wie zu Beginn der **asm**-Anweisung. Die Register AX, BX, CX, DX, SI, DI, ES und das Flag-Register stehen dagegen ohne Einschränkung zur Verfügung. Beim Eintritt in einen Assembler-Block können Sie von folgendem ausgehen: BP zeigt auf den aktuellen Stack-Rahmen, SP zeigt auf die Spitze des Stack, SS enthält die Segmentadresse des Stack-Segments und DS die Segmentadresse des Daten-Segments. Die anderen Register haben keinen definierten Inhalt.

Die Syntax einer Assembler-Anweisung

Assembler-Anweisungen haben folgende Syntax:

```
[ Label ":" ] <Präfix> [ Opcode [Operand <" , " Operand > ] ]
```

Label stellt einen Label-Bezeichner dar, *Präfix* ist ein Assemblerpräfixopcode (Operationscode), *Opcode* ein Assembler-Befehl oder eine Anweisung, *Operand* ist gültiger Assembler-Ausdruck.

Kommentare sind zwischen den einzelnen Assembler-Anweisungen erlaubt, jedoch nicht innerhalb einer Anweisung. Zulässig ist beispielsweise folgendes:

```
asm
    MOV  AX,1      {Startwert}
    MOV  CX,100    {Zähler}
end
```

Das nächste Programmfragment dagegen ist fehlerhaft:

```
asm
    MOV  {Startwert} AX,1
    MOV  CX, {Zähler} 100
end
```

Labels

Labels sehen in Assembler genauso aus wie in Pascal: ein Label-Bezeichner gefolgt von einem Doppelpunkt. Genau wie in Pascal muß ein Label, das in einer **asm**-Anweisung definiert ist, im **label**-Deklarationsteil des betreffenden Programmabschnittes deklariert sein. Von dieser Regel gibt es nur eine Ausnahme: *lokale Labels*.

Lokale Labels beginnen mit einem Klammeraffen (@) und sind, da Pascal-Bezeichner keinen Klammeraffen enthalten dürfen, automatisch auf die Verwendung innerhalb von **asm**-Anweisungen beschränkt. Der Gültigkeitsbereich erstreckt sich vom Schlüsselwort **asm** bis zum zugehörigen **end**.

Hinweis: Lokale Labels müssen, ganz wie in Turbo Assembler, nicht explizit definiert werden.

Der Bezeichner eines lokalen Labels besteht aus einem Klammeraffen (@) gefolgt von einem oder mehreren Buchstaben (A..Z), Ziffern (0..9), Unterstrichen (_) oder weiteren Klammeraffen (@).

Der folgende Programmausschnitt zeigt die Verwendung normaler und lokaler Labels in **asm**-Anweisungen:

```
labelStart, Stop
...
begin
    asm
        Start:
        ...
        JZ    Stop
```

```

@1:
    ...
    LOOP    @1
end;
asm
@1:
    ...
    JC      @2
    ...
    JMP     @1
@2:
end;
goto Start;
Stop:
end;

```

Wie dieser Programmausschnitt zeigt, kann ein normales Label innerhalb einer **asm**-Anweisung definiert werden. Verweise auf normale Labels können sowohl von außerhalb einer **asm**-Anweisung, wie auch von innerhalb erfolgen. Wie Sie weiter sehen, ist es zulässig, in verschiedenen **asm**-Anweisungen gleichlautende Bezeichner für lokale Labels zu benutzen, da ihr Gültigkeitsbereich ohnehin auf die **asm**-Anweisung beschränkt ist, in der sie verwendet werden.

Präfix-Opcodes

Der integrierte Assembler unterstützt folgende Präfixe:

LOCK	Bus lock
REP	wiederhole String-Operation
REPE/REPZ	wiederhole String-Operation solange Equal- bzw. Zero-Flag gesetzt ist
REPNE/REPNZ	wiederhole String-Operation solange Equal- bzw. Zero-Flag nicht gesetzt ist
SEGCS	CS-Register-Vorgabe
SEGDS	DS-Register-Vorgabe
SEGES	ES-Register-Vorgabe
SEGSS	SS-Register-Vorgabe

Ein Assembler-Befehl wird, soweit überhaupt, nur selten mit mehr als einem einzigen Präfix eingeleitet. Sinnvolle Kombinationen mit mehr als drei Präfixen (wie z.B. *LOCK SEGxx REPxx*) sind ohnehin kaum vorstellbar. Auf alle Fälle sollte man bei der Verwendung mehrerer Präfixe sehr vorsichtig sein (wichtig ist z.B. ihre Reihenfolge), da einige Prozessoren der 80x86-Familie manche Präfix-Kombinationen nicht korrekt verarbeiten. Tritt beispielsweise während einer String-Operation mit vorangestelltem *REP*-Präfix ein Interrupt auf, so arbeitet die CPU die wiederholte String-Operation auch nach Beendigung des Interrupt korrekt ab. Tritt der Interrupt dagegen zu einem Zeitpunkt auf, wo zwar das *REP*-Präfix, der String-Befehl dagegen noch nicht eingelesen wurde, so »vergißt« ein 8088/8086-Prozessor das vorangegangene *REP*-Präfix.

Wie Sie sehen, ist es schon aus diesem Grund eine äußerst unsichere Sache,

ein *LOCK*- und/oder ein *SEGxx*-Präfix zusammen mit einem *REP*-Präfix zu verwenden.

Hier einige Beispiele zur Verwendung von Präfixen:

```
asm
  REP    MOVSB      { kopiere CX Bytes von DS:SI nach ES:DI }
  SEGES  LODSW      { lade word von ES:SI }
  SEGCS  MOV AX,[BX] { identisch mit: MOV AX,CS:[BX] }
  SEGES                      { bezieht sich auf die nächste Anweisung }
  MOV WORD PTR [DI],0 { wird zu: MOV WORD PTR ES:[DI],0 }
end;
```

Beachten Sie, daß ein Präfix in einer Anweisung auch ohne Assembler-Befehl vorkommen kann. In einem solchen Fall bezieht sich das Präfix auf den Assembler-Befehl der darauf folgenden Anweisung.

Befehlssätze

Der integrierte Assembler unterstützt alle 8086/8087- und 80286/80287-Befehle. Für den Einsatz der 8087-Befehle muß der Compiler-Schalter für den numerischen Koprozessor {\$N+} eingeschaltet werden, die 80286-Befehle werden durch den Compiler-Schalter für den 80286-Prozessor {\$G+} aktiviert. Sind beide Compiler-Schalter {\$G+,N+} eingeschaltet, so sind damit auch die Befehle des 80287-Koprozessors verfügbar.

Eine komplette Beschreibung aller Assembler-Befehle finden Sie im Referenzhandbuch von Turbo Assembler 2.0 sowie in den Referenzbüchern der 80x86- und 80x87-Prozessoren.

Automatische Anpassung des RET-Befehls Der *RET*-Befehl erzeugt, in Abhängigkeit vom Aufrufmodell der aktuellen Funktion oder Prozedur, für den Rücksprung automatisch einen Near- oder Far-Maschinencode.

```
procedure NearProc; near;
begin
  asm
    RET    { erzeugt einen Near-Rücksprung }
  end;
end;

procedure FarProc; far;
begin
  asm
    RET    { erzeugt einen Far-Rücksprung }
  end;
end;
```

Die Befehle *RETN* bzw. *RETF* erzeugen dagegen, unabhängig vom Aufrufmodell, gezielt einen Near- und Far-Rücksprung.

Automatische Sprunganpassung Solange nicht anders spezifiziert, optimiert der integrierte Assembler Sprung-Befehle durch Einsetzen der kürze-

sten und damit effizientesten Befehlsvariante. Diese Optimierung erfolgt bei allen unbedingten Sprüngen (*JMP*), aber auch bei allen bedingten Sprüngen, wenn als Sprungziel ein Label, nicht jedoch eine Funktion oder Prozedur angegeben ist.

Liegt das Ziel-Label eines unbedingten Sprunges innerhalb einer Distanz von -128 bis +127 Bytes, dann generiert der integrierte Assembler einen kurzen Sprung (ein Byte Opcode und ein Byte Distanz); andernfalls wird ein Near-Sprung, bestehend aus einem Byte Opcode und zwei Bytes Distanz erzeugt.

Auch für einen bedingten Sprung wird ein kurzer Sprung generiert, wenn das Ziel-Label innerhalb einer Distanz von -128 bis +127 Bytes entfernt ist. Liegt das Sprungziel jedoch außerhalb dieses Bereiches, generiert der integrierte Assembler zwei Sprungbefehle: einen kurzen Sprung mit invertierter Sprungbedingung über einen darauffolgenden Near-Sprung, der zum eigentlichen Sprungziel führt. Diese Kombination besteht somit aus zwei Assembler-Befehlen und hat eine Länge von fünf Bytes. Für das Beispiel

```
JC      Stop
```

erzeugt der integrierte Assembler folgende Sequenz (wobei *Stop* außerhalb der für kurze Sprünge erreichbaren Distanz liegt):

```
JNC     Skip  
JMP     Stop  
Skip:
```

Sprünge auf Funktionen und Prozeduren werden immer entweder als Near oder Far erzeugt, niemals aber als kurze (short) Sprünge. Außerdem erlaubt der integrierte Assembler keine bedingten Sprünge auf Funktionen oder Prozeduren. Möglich ist es hingegen, über die Konstrukte *NEAR PTR* bzw. *FAR PTR*, gezielt einen Near- oder Far-Sprung auf ein Label zu erzwingen, selbst wenn das Sprungziel durch einen kurzen Sprung erreichbar wäre. Zum Beispiel wird, ungeachtet der tatsächlichen Sprungdistanz, für

```
JMP     NEAR PTR Stop  
JMP     FAR PTR Stop
```

immer ein Near-Sprung bzw. ein Far-Sprung erzeugt.

Assembler-Anweisungen

Der integrierte Assembler von Turbo Pascal unterstützt drei verschiedene Anweisungen zur Erzeugung von Datenstrukturen: *DB* (define byte), *DW* (define word) und *DD* (define double word). Diese drei Anweisungen erzeugen Daten, entsprechend der nachfolgenden, durch Kommata voneinander getrennten Operanden.

Die *DB*-Anweisung generiert eine Folge von Daten-Bytes. Jeder Operand muß ein konstanter Ausdruck zwischen -128 und 255 oder eine Zeichenkette beliebiger Länge sein. Konstanten-Ausdrücke erzeugen Code von einem Byte Länge, Zeichenketten ergeben eine Byte-Folge, wobei der Wert eines jeden Bytes dem ASCII-Code des angegebenen Zeichens entspricht.

Die *DW*-Anweisung generiert eine Folge von Daten-Words. Jeder Operand muß ein konstanter Ausdruck zwischen -32768 und 65535 oder ein Adreß-Aus-

druck sein. Im Falle eines Adreß-Ausdruckes erzeugt der integrierte Assembler ein Word, dessen Inhalt einem Near-Zeiger, also dem Offset-Anteil der angegebenen Adresse, entspricht.

Die *DD*-Anweisung generiert eine Folge von Double-Words. Jeder Operand muß ein konstanter Ausdruck zwischen -2147483648 und 4294967295 oder ein Adreß-Ausdruck sein. Im Falle eines Adreß-Ausdruckes erzeugt der integrierte Assembler einen Far-Zeiger; d.h. es wird ein vollständiger 32-Bit-Zeiger mit Offset- und Segment-Anteil abgelegt.

Solchermaßen erzeugte Daten werden, genauso wie auch der vom integrierten Assembler generierte Programmcode, immer im Code-Segment abgelegt. Im Daten-Segment werden auch weiterhin nur diejenigen initialisierten oder uninitialisierten Daten abgelegt, die über eine entsprechende Pascal-Deklaration (**const** oder **var**) deklariert wurden.

Hier einige Beispiele für die verschiedenen Anweisungen:

```
asm
DB      0FFH                { ein Byte }
DB      0, 99                { zwei Bytes }
DB      'A'                  { Ord(A) }
DB      'Hallo Welt...', 0DH, 0AH { String und CR/LF }
DB      12, "Turbo Pascal"   { String wie in Pascal }
DW      0FFFFH               { ein Word }
DW      0, 9999              { zwei Words }
DW      'A'                  { entspricht DB 'A', 0 }
DW      'BA'                 { entspricht DB 'A', 'B' }
DW      MeineVar             { Offset von MeineVar }
DW      MeineProc            { Offset von MeineProc }
DD      0FFFFFFFFH           { ein Double-Word }
DD      0, 9999999999        { zwei Double-Words }
DD      'A'                  { entspricht
                             DB 'A', 0, 0, 0 }
DD      'DCBA'               { entspricht
                             DB 'A', 'B', 'C', 'D' }
DD      MeineVar             { Zeiger auf MeineVar }
DD      MeineProc            { Zeiger auf MeineProc }
end;
```

In Turbo Assembler wird durch das Vorstellen eines Bezeichners vor eine *DB*-, *DW*- oder *DD*-Anweisung eine Variable entsprechender Größe an der Speicherstelle der Anweisung deklariert. Turbo Assembler erlaubt z.B. folgendes:

```
ByteVar  DB  ?
WordVar  DW  ?
...
MOV      AL, ByteVar
MOV      BX, WordVar
```

Der integrierte Assembler unterstützt solche Deklarationen nicht, die einzigen Symbole, die man innerhalb einer **asm**-Anweisung definieren kann, sind Labels. Variablen müssen auch künftig gemäß der Pascal-Syntax im Pascal-

Deklarationsteil deklariert werden. Das eben gezeigte Beispiel wird im integrierten Assembler folgendermaßen formuliert:

```
var
  ByteVar:   Byte;
  WordVar:   Word;
...
asm
  MOV    AL, ByteVar
  MOV    BX, WordVar
end;
```

Operanden

Operanden sind Ausdrücke, bestehend aus einer Kombination von Konstanten, Registern, Symbolen und Operatoren. Obwohl die Syntax des integrierten Assemblers in vielen Fällen der Pascal-Syntax angelehnt wurde, gibt es doch eine Reihe von Unterschieden, die im folgenden erklärt werden.

Nachstehende reservierte Wörter haben, soweit sie als Operanden vorkommen, für den integrierten Assembler eine vordefinierte Bedeutung:

AH	CH	DX	OR	ST
AL	CL	ES	PTR	TBYTE
AND	CS	FAR	QWORD	TYPE
AX	CX	HIGH	SEG	WORD
BH	DH	LOW	SHL	XOR
BL	DI	MOD	SHR	
BP	DL	NEAR	SI	
BX	DS	NOT	SP	
BYTE	DWORD	OFFSET	SS	

Diese reservierten Wörter haben immer Vorrang vor selbstdefinierten Bezeichnern. So wird zum Beispiel im folgenden Code-Ausschnitt

```
var
  CH:   Char;
...
asm
  MOV    CH, 1
end;
```

der Wert 1 *nicht* der Variablen CH, sondern dem Register CH zugewiesen. Um die Variable CH anzusprechen, muß dem Variablennamen das Et-Zeichen (&) vorangestellt werden:

```
asm
  MOV    &CH, 1
end;
```

Vermeiden Sie möglichst Bezeichnernamen, die mit reservierten Wörtern identisch sind, denn eine Verwechslung von Bezeichnern führt sehr leicht zu kaum auffindbaren Fehlern.

Ausdrücke

Ausdrücke werden im integrierten Assembler immer als 32-Bit-Werte berechnet. Gleitkomma-Zahlen und Strings, ausgenommen String-Konstanten, werden nicht akzeptiert.

Die Ausdrücke des integrierten Assemblers setzen sich aus *Ausdruck-Elementen* und *Operatoren* zusammen. Außerdem hat jeder Ausdruck eine *Ausdruck-Klasse* und einen *Ausdruck-Typ*. Dieses Konzept wird in den folgenden Abschnitten noch näher erläutert.

Unterschiede zwischen Pascal- und Assembler-Ausdrücken

Der wichtigste Unterschied zwischen Ausdrücken in Turbo Pascal und Ausdrücken im integrierten Assembler besteht darin, daß Ausdrücke im integrierten Assembler stets einen *konstanten Wert* haben müssen. Mit anderen Worten: der Wert eines Ausdrucks muß zum Zeitpunkt der Compilierung feststehen. Das folgende Beispiel soll dies verdeutlichen. Wenn

```
const
  X = 10;
  Y = 20;
var
  Z : Integer;
```

eine gegebene Deklaration ist, ergibt folgende Zeile eine gültige Assembler-Anweisung:

```
asm
  MOV    Z,X+Y
end;
```

Da sowohl *X* als auch *Y* Konstanten sind, ist es faktisch gleichgültig, ob im letzten Beispiel der Ausdruck *X+Y*, oder stattdessen die Konstante 30 angegeben ist. Der Ausdruck *X+Y* stellt lediglich eine zweckdienlichere Schreibweise ein und derselben Konstanten dar. Der aus dieser Anweisung resultierende Maschinen-Befehl ist ein Ladebefehl, durch den die Variable *Z* mit dem Wert 30 geladen wird. Verwendet man anstelle der Konstanten *X* und *Y* allerdings Variablen, wird der Unterschied zwischen Pascal und Assembler deutlich:

```
var
  X, Y : Integer;
```

Die Summe aus *X* und *Y* ist zum Zeitpunkt der Compilierung nicht bekannt. Der Assembler-Block zur Berechnung der Summe aus *X+Y* muß daher wie folgt geändert werden:

```
asm
  MOV    AX,X
  ADD    AX,Y
  MOV    Z,AX
end;
```

Ein weiterer wichtiger Unterschied zwischen Pascal und Ausdrücken im integrierten Assembler betrifft die Interpretation von Variablen. Während sich eine Variablenreferenz in Pascal immer auf den *Inhalt* der Variablen bezieht, betrachtet der integrierte Assembler eine Variablenreferenz als eine Referenz auf die *Adresse* der Variablen. Ein Ausdruck wie $X+4$ stellt in Pascal die Summe des Inhalts der Variablen X und der Konstanten 4 dar. In Assembler bezieht sich der Ausdruck $X+4$ dagegen auf den Inhalt des Word an der Speicheradresse, die vier Bytes höher liegt als die Adresse der Variablen X . Mit folgender, erlaubter Assembler-Anweisung

```
asm
    MOV    AX,X+4
end;
```

wird das AX-Register also *nicht* mit der Summe $X+4$ geladen, sondern mit dem Inhalt des Word, das vier Bytes oberhalb von X gespeichert ist. Die korrekte Anweisung für die Addition sieht so aus:

```
asm
    MOV    AX,X
    ADD    AX,4
end;
```

Ausdruck-Elemente

Die grundlegenden Elemente eines Ausdrucks sind *Konstanten*, *Register* und *Symbole*.

Konstanten Der integrierte Assembler unterstützt zwei verschiedene Arten von Konstanten: *numerische Konstanten* und *String-Konstanten*.

Numerische Konstanten: Numerische Konstanten sind ganze Zahlen im Wertebereich von 2147483649 bis 4294967295.

Gemäß Standardeinstellung werden numerische Konstanten in dezimaler Schreibweise geschrieben (Basis 10). Der integrierte Assembler unterstützt allerdings auch die binäre (Basis 2), die oktale (Basis 8) und die hexadezimale (Basis 16) Schreibweise. Die Wahl der Zahlenbasis erfolgt durch das Anhängen eines Suffix an die betreffende Zahl: Suffix *B* für binär, Suffix *O* für oktäl und Suffix *H* (bzw. Voranstellen des Präfix $\$$, wie in Pascal-Ausdrücken) für hexadezimal.

Vorsicht! Für Pascal-Ausdrücke bleibt alles beim Alten: Zur Verfügung stehen ausschließlich das dezimale und das hexadezimale Zahlensystem; Hexadezimal-Zahlen werden immer mit dem Präfix $\$$ eingeleitet.

Numerische Konstanten müssen mit einer Ziffer von 0 bis 9 beginnen bzw. mit dem Präfix $\$$ bei hexadezimalen Zahlen. Verwendet man bei hexadezimalen Zahlen das Suffix H, muß auch diese Zahl mit einer Ziffer von 0 bis 9 beginnen (z.B. 0F5Fh). Diese Schreibweise ist notwendig, um hexadezimele Zahlen von Bezeichnern unterscheiden zu können.

String-Konstanten: String-Konstanten müssen entweder von Apostrophen ('...') oder Anführungszeichen ("...") eingeschlossen werden. Soll ein den String einschließendes Zeichen selbst Bestandteil des Strings sein, so muß es innerhalb

des Strings doppelt angegeben werden. Hier einige Beispiele von String-Konstanten:

```
'Z'
'Turbo Pascal'
'Das war's'
'"Das war''s", sagte er.'
'100'
'''
'''
```

Beachten Sie, daß in der vierten Zeichenkette zwei aufeinanderfolgende Apostrophen verwendet werden, um einen einzigen Apostrophen als String-Bestandteil zu erzeugen.

String-Konstanten, die über *DB*-Anweisungen deklariert sind, haben keine Längenbeschränkung und ergeben eine Byte-Folge, welche den ASCII-Werten der einzelnen Zeichen entspricht. Werden String-Konstanten allerdings mit den Anweisungen *DW* oder *DD* erzeugt, dann dürfen diese Konstanten eine Länge von zwei bzw. vier Bytes nicht überschreiten. Die beiden letzten Anweisungen bezeichnen einen numerischen Wert, der auch Teil eines Ausdruckes sein kann und sich wie folgt berechnen läßt:

```
Ord(Ch1) + Ord(Ch2) shl 8 + Ord(Ch3) shl 16 + Ord(Ch4) shl 24
```

Z1 entspricht dem letzten Zeichen, *Z4* dem ersten Zeichen der Zeichenkette. Ist die Zeichenkette kürzer als vier Zeichen, werden die entsprechenden fehlenden Bytes auf null gesetzt. Hier einige Beispiele von String-Konstanten und deren numerischer Repräsentation:

```
'a'      00000061H
'ba'     00006261H
'cba'    00636261H
'dcba'   64636261H
'a '     00006120H
'a'*2    000000E2H
'a'-'A'  00000020H
not 'a'  FFFFFFF9H
```

Register Die folgenden reservierten Symbole bezeichnen die CPU-Register:

allgemeine 16-Bit-Register	AX	BX	CX	DX
niederwertige 8-Bit-Register	AL	BL	CL	DL
höherwertige 8-Bit-Register	AH	BH	CH	DH
16-Bit-Zeiger-Register	SP	BP	SI	DI
16-Bit-Segment-Register	CS	DS	SS	ES
80x87-Register-Stack	ST			

Wenn ein Operand aus nur einem Register-Bezeichner besteht, nennt man diesen Operanden einen Register-Operanden. Alle Register können als Register-Operanden dienen. Darüber hinaus können manche Register noch in einem anderen Zusammenhang verwendet werden.

- Die Basis-Register (BP und BX), ebenso wie die Index-Register (SI und DI) können – in eckigen Klammern geschrieben – für indizierte Speicherzugriffe verwendet werden. Folgende Kombinationen von Basis- und Index-Registern sind erlaubt: [BX], [BP], [SI], [DI], [BX+SI], [BX+DI], [BP+SI], [BP+DI].
- Die Segment-Register (ES, CS, SS und DS) – gefolgt von einem Doppelpunkt – können zur Überschreibung der Segmentvorgabe benutzt werden.
- Das Symbol *ST* bezeichnet die Spitze des Koprozessor-Register-Stack. Jedes der acht Koprozessor-Register kann durch das Symbol *ST* gefolgt von der Registernummer in runden Klammern (*ST(x)*) angesprochen werden. Die Register sind von 0 bis 7 nummeriert, wobei die Nummer die Distanz zur Spitze des Register-Stack angibt.

Symbole Der integrierte Assembler ermöglicht den Zugriff auf fast alle Pascal-Symbole. Diese Symbole können Labels, Konstanten, Typ-Bezeichner, Variablen, Prozeduren und Funktionen sein. Zusätzlich stellt der integrierte Assembler noch folgende spezielle Symbole zu Verfügung:

@Code @Data @Result

Die Symbole *@Code* und *@Data* repräsentieren das aktuelle Code- bzw. Daten-Segment. Sie sollten nur zusammen mit dem *SEG*-Operator verwendet werden:

```
asm
  MOV  AX, SEG @Data
  MOV  DS, AX
end;
```

Über das Symbol *@Result* können Sie innerhalb des Anweisungsteils einer Funktion auf das Ergebnis dieser Funktion zugreifen. In der Funktion

```
function Sum (X, Y: Integer): Integer;
begin
  Sum := X + Y;
end;
```

wird der Wert des Funktionsergebnisses der Variablen *Sum* zugewiesen. Wäre diese Funktion in Assembler geschrieben, müßten Sie das Funktionsergebnis über *@Result* zurückliefern:

```
function Sum (X, Y: Integer): Integer;
begin
  asm
    MOV  AX, X
    ADD  AX, Y
    MOV  @Result, AX
  end;
end;
```

In diesem Beispiel mag die Verwendung des *@Result*-Symbols überflüssig erscheinen, da der Rückgabewert einer Integer-Funktion sowieso im AX-Register

erwartet wird. Bei normalen Funktionen wird jedoch im Eintrittscode der Funktion Platz für das Funktionsergebnis auf dem Stack reserviert. Beim Austritt wird der Inhalt dann in die entsprechenden Register kopiert und steht dem Aufrufer zur Verfügung. Mit anderen Worten ist die Verwendung des *@Result*-Symbols absolut notwendig (vergl. Assembler-Funktionen). Folgende Symbole können nicht in Assembler-Ausdrücken verwendet werden:

- Standard-Prozeduren und -Funktionen (z.B. *WriteLn*, *Chr*)
- Die vordefinierten Pseudo-Arrays *Mem*, *MemW*, *MemL*, *Port* und *PortW*
- String-, Gleitkomma- und Mengen-Konstanten
- Prozeduren und Funktionen, die mit der **Inline**-Anweisung deklariert sind
- Labels, die nicht im aktuellen Block definiert sind
- Das Symbol *@Result* außerhalb von Funktionen

Tabelle 15.1 enthält eine Zusammenfassung der Werte, Klassen und Typen der verschiedenen Arten von Symbolen, die in Assembler-Ausdrücken zulässig sind (Ausdruck-Klassen und -Typen werden in einem der folgenden Abschnitten behandelt).

Tabelle 15.1 *Symbol-Werte, -Klassen und -Typen*

Symbol	Wert	Klasse	Typ
Label	Label-Adresse	Speicher	Short
Konstante	Konstantenwert	Konstante	0
Type	0	Speicher	wie <i>SizeOf(Type)</i>
Feld	Feld-Offset	Speicher	wie <i>SizeOf(Feld)</i>
Variable	Variablen-Adresse	Speicher	wie <i>SizeOf(Variable)</i>
Prozedur	Prozedur-Adresse	Speicher	NEAR oder FAR
Funktion	Funktion-Adresse	Speicher	NEAR oder FAR
Unit	0	Konstante	0
@Code	Adresse des Code-Segments	Speicher	0FFF0H
@Data	Adresse des Daten-Segments	Speicher	0FFF0H
@Result	Offset der Result-Variablen	Speicher	

Lokale Variablen (Variablen, die in Funktionen und Prozeduren deklariert sind) werden immer auf dem Stack angelegt und relativ zu *SS:BP* adressiert. Der Wert des Symbols einer lokalen Variablen gibt den vorzeichenbehafteten Offset relativ zu *SS:BP* an. Bei Referenzen auf lokale Variablen fügt der Assembler automatisch den Bezug auf das Register *BP* hinzu. So übersetzt der integrierte Assembler bei folgender Deklaration

```
procedure Test;  
var  
    Count: Integer;
```

die **asm**-Anweisung

```
asm
  MOV  AX, Count
end;
```

in folgenden Assembler-Befehl:

```
MOV  [BP-2]
```

Der integrierte Assembler behandelt **var**-Parameter immer als 32-Bit-Zeiger mit einer Größe von vier Bytes. In Pascal gibt es keinen Unterschied zwischen dem Zugriff auf **var**-Parameter und dem auf Werteparameter. Im integrierten Assembler ist das anders. Da es sich bei **var**-Parametern tatsächlich um Zeiger handelt, müssen sie im integrierten Assembler auch als solche behandelt werden. Wenn Sie also auf den Inhalt eines **var**-Parameters zugreifen möchten, müssen Sie zunächst den 32-Bit-Zeiger laden und dann auf die Speicherstelle zugreifen, auf die der Zeiger verweist. Wären die Parameter *X* und *Y* aus der obigen Funktion *Sum* nun **var**-Parameter, ergäbe dies folgenden Programm-Code:

```
function Sum (var X, Y: Integer): Integer;
begin
  asm
    LES  BX,X
    MOV  AX,ES:[BX]
    LES  BX,Y
    ADD  AX,ES:[BX]
    MOV  @Result,AX
  end;
end;
```

Manche Symbole, wie beispielsweise Record-Typen und -Variablen, markieren einen Speicherbereich, dessen Felder mit einem Punkt und dem Feldbezeichner angesprochen werden können. Bei der folgenden Deklaration

```
type
  Point = record
    X, Y: Integer;
  end;
  Rect = record
    A, B: Point;
  end;

var
  P: Point;

  R: Rect;
```

können die Felder der Variablen *P* und *R* wie folgt angesprochen werden:

```
asm
  MOV  AX,P.X
  MOV  DX,P.Y
  MOV  CX,R.A.X
```

```
MOV BX,R.B.Y
end;
```

Typ-Bezeichner können bei Speicheroperationen zur Erzeugung des richtigen Offsets eingesetzt werden. Für die folgenden fünf Anweisungen wird ein und derselbe Maschinen-Code erzeugt, der den Inhalt der Speicherstelle ES:[DI+4] in das AX-Register lädt.

```
asm
MOV AX,(Rect PTR ES:[DI]).B.X
MOV AX,Rect(ES:[DI]).B.X
MOV AX,ES:Rect[DI].B.X
MOV AX,Rect[ES:DI].B.X
MOV AX,ES:[DI].Rect.B.X
end;
```

Der Speicherbereich eines Record- bzw. Objekt-Typs wird von dessen Typ-, Feld- und Variablen-Symbolen beschrieben. Zusätzlich kann man mit einem Unit-Bezeichner, in gleicher Weise wie in Pascal, auf ein Symbol dieser Unit zugreifen.

Ausdruck-Klassen

Der integrierte Assembler kennt drei unterschiedliche Klassen von Ausdrücken: *Register-Ausdrücke*, *Speicher-Referenzen* und *konstante Ausdrücke*.

Ein Ausdruck, der nur aus einem Registernamen besteht, ist ein Register-Ausdruck. Beispiele für Register-Ausdrücke sind *AX*, *CL*, *DI* und *ES*. Register-Ausdrücke, die als Operanden verwendet werden, weisen den Assembler an, Befehle zu generieren, die mit den CPU-Registern operieren.

Ausdrücke, die auf Speicherstellen verweisen, sind Speicher-Referenzen; zu dieser Kategorie gehören Pascal-Labels, Variablen, typisierte Konstanten, Prozeduren und Funktionen.

Ausdrücke, die weder Register- noch Speicher-Ausdrücken angehören, sind konstante Ausdrücke; in diese Gruppe gehören untypisierte Konstanten und Typ-Bezeichner.

Werden konstante Ausdrücke und Speicher-Ausdrücke als Operanden verwendet, dann wird für sie unterschiedlicher Code erzeugt.

```
const
Start = 10;

var
Count: Integer;

...
asm
MOV AX,Start      {MOV AX,xxxx}
MOV BX,Count      {MOV BX,[xxxx]}
MOV CX,[Start]    {MOV CX,[xxxx]}
MOV DX,OFFSET Count {MOV DX,xxxx}
end;
```

Da es sich bei *Start* um einen konstanten Ausdruck handelt, erzeugt der Assembler einen Befehl, der das Register mit einem konstanten Wert lädt (move immediate). Die zweite *MOV*-Anweisung dagegen wird in eine Anweisung für einen Speicherzugriff (move memory) übersetzt, da *Count* eine Speicher-Referenz darstellt. Bei der dritten *MOV*-Anweisung bewirken die eckigen Klammern, daß *Start* als eine Speicher-Referenz behandelt wird (Inhalt der Speicherstelle mit Offset 10). In der vierten *MOV*-Anweisung schließlich wird *Count* mittels des vorangestellten *OFFSET*-Operators in einen konstanten Ausdruck (Offset von *Count* im Daten-Segment) verwandelt.

Wie Sie sehen, haben der *OFFSET*-Operator und die eckigen Klammern eine zueinander komplementäre Wirkung. Der aus den beiden folgenden Assembler-Befehlen resultierende Maschinen-Code, ist daher identisch mit dem Maschinen-Code der beiden ersten Assembler-Befehle aus dem letzten Beispiel:

```
asm
    MOV  AX,OFFSET [Start]
    MOV  BX,[OFFSET Count]
end;
```

Speicher-Ausdrücke und konstante Ausdrücke unterteilen sich zusätzlich in *relozierbare Ausdrücke* und *absolute Ausdrücke*. Ein relozierbarer Ausdruck erfordert noch eine *Relokation* während des Link-Vorgangs, ein absoluter Ausdruck hingegen nicht. Ausdrücke, die sich auf Prozeduren, Funktionen, Variablen und Labels beziehen, sind relozierbare Ausdrücke. Beinhaltet ein Ausdruck dagegen ausschließlich einen konstanten Wert, so handelt es sich um einen absoluten Ausdruck.

Relokation: Zur Übersetzungszeit ist dem Compiler die endgültige Adresse einer Prozedur, Funktion, Variablen oder eines Labels nicht bekannt. Dieser Zustand ändert sich erst, wenn der Linker Symbolen eine spezifische absolute Adresse zuweist.

Der integrierte Assembler erlaubt beliebige Operationen mit absoluten Ausdrücken, begrenzt aber die Operationen mit relozierbaren Ausdrücken auf die Addition bzw. Subtraktion von Konstanten.

Ausdruck-Typen

Jeder Ausdruck des integrierten Assemblers hat einen bestimmten Typ oder, genauer gesagt, eine bestimmte Größe, da der Assembler den Typ eines Ausdrucks als eine Größenangabe für den Speicherzugriff interpretiert. Zum Beispiel belegt eine *Integer*-Variable zwei Bytes im Speicher und demzufolge hat der Betrag ihrer Größe (ihres Typs) den Wert zwei.

Der integrierte Assembler führt Typ-Prüfungen durch, wo immer dies möglich ist. In den folgenden Anweisungen

```
var
    QuitFlag: Boolean;
    OutBufPtr: Word;
    ...
asm
    MOV  AL,QuitFlag
```

```
MOV BX,OutBufPtr
end;
```

prüft der Assembler, ob die Größe von *QuitFlag* ein Byte und die Größe von *OutBufPtr* zwei Bytes beträgt. Stellt der Assembler unterschiedliche Typen (Größen) fest, resultiert daraus eine Fehlermeldung. Folgendes ist zum Beispiel nicht erlaubt

```
asm
MOV DL,OutBufPtr
end;
```

da es sich bei DL um ein 8-Bit-Register (1 Byte) handelt, *OutBufPtr* jedoch eine Größe von zwei Bytes hat. Allerdings kann der Typ eines Speicher-Ausdrucks durch entsprechende Operatoren geändert werden. Korrekte Schreibweisen für das letzte Beispiel sind

```
asm
MOV DL,BYTE PTR OutBufPtr
MOV DL,BYTE (OutBufPtr)
MOV DL,OutBufPtr.Byte
end;
```

wobei sich alle Zugriffe auf das erste (niederwertige) Byte von *OutBufPtr* beziehen.

In manchen Fällen ist ein Speicher-Ausdruck untypisiert. Ein Beispiel dafür ist ein konstanter Ausdruck in eckigen Klammern:

```
asm
MOV AL,[100H]
MOV BX,[100H]
end;
```

Der integrierte Assembler erlaubt beide Anweisungen, da der Ausdruck [100H] nicht typisiert ist. Der Ausdruck [100H] stellt lediglich einen Bezug auf den Inhalt der Adresse 100H im Daten-Segment dar, der Ausdruck-Typ ist durch das verwendete Register (AL für das Byte und BX für das Word) bestimmt. Situationen, in denen der Assembler keine Typ-Feststellung über einen zweiten Operator durchführen kann, erfordern eine explizite Typangabe:

```
asm
INC BYTE PTR [100H]
IMUL WORD PTR [100H]
end;
```

Tabelle 15.2 faßt die vordefinierten Typ-Symbole zusammen, die der integrierte Assembler zusätzlich zu den in Pascal deklarierten zur Verfügung stellt.

Tabelle 15.2 vordefinierte Typ-Symbole

Symbol	Typ
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10
NEAR	0FFFFH
FAR	0FFFEH

Beachten Sie insbesondere die Pseudo-Typen **NEAR** und **FAR**, die bei Prozedur- und Funktions-Symbolen zur Angabe des Aufrufmodells dienen. Ebenso wie andere Symbole auch, können Sie in Typangaben **NEAR** und **FAR** verwenden. *FarProc* sei zum Beispiel eine **FAR**-Prozedur:

```
procedure FarProc; far;
```

Wenn Sie nun im selben Modul Assembler-Code schreiben, ist es möglich, die weit effizientere **NEAR**-Anweisung für den Prozedur-Aufruf einzusetzen:

```
asm
    PUSH CS
    CALL NEAR PTR FarProc
end;
```

Ausdruck-Operatoren

Der integrierte Assembler verfügt über eine Vielzahl von Operatoren. Die Rangfolge dieser Operatoren unterscheidet sich jedoch erheblich von der Rangfolge in Pascal. So hat zum Beispiel der **AND**-Operator einen niedrigeren Rang als die Plus(+) und Minus(-) Operatoren; für Pascal gilt genau das Gegenteil: hier hat der **AND**-Operator Vorang. Im folgenden werden die Operatoren des integrierten Assemblers gemäß ihrer Priorität in absteigender Reihenfolge aufgelistet.

& »Bezeichner-Vorgabe« – der Bezeichner nach dem Et-Zeichen wird als benutzerdefiniertes Symbol behandelt, selbst wenn ein vordefiniertes Symbol gleichen Namens im integrierten Assembler existiert.

(. . .) »Teilausdrücke« – Ausdrücke innerhalb runder Klammern werden komplett ausgewertet und als einziges Ausdruck-Element behandelt. Ein weiterer Ausdruck kann diesem Teilausdruck vorangestellt werden. Das Ergebnis ergibt sich aus der Summe beider Ausdrücke, wobei der Ergebnis-Typ dem Typ des vorangestellten Ausdruckes entspricht.

[. . .] »Speicher-Referenzen« – der Ausdruck innerhalb der eckigen Klammern wird komplett ausgewertet und als einzelnes Ausdruck-Element behandelt. Der Klammerausdruck kann für indizierte Speicherzugriffe eine Kombination der Register *BX*, *BP*, *SI* oder *DI* und des Plus (+) bzw. Minus (-) Operators enthalten. Ist dem Klammerausdruck ein weiterer Ausdruck vorangestellt, ergibt sich das

Ergebnis aus der Summe beider Ausdrücke. Das Ergebnis ist immer eine Speicher-Referenz und hat den gleichen Typ wie der vorangestellte Ausdruck.

. »Strukturauswahl« – dieser Operator findet, genau wie in Pascal, beim Zugriff auf Felder eines Records Anwendung. Der Typ des Gesamtausdruckes entspricht dem Typ des Ausdruckes nach dem Punkt, also dem Typ des Feldes auf das der Zugriff erfolgt.

HIGH Liefert die acht höherwertigen Bits des konstanten Ausdruckes, der dem Operator folgt, zurück. Die Größe des Ausdruckes muß dem Datentyp Word entsprechen.

LOW Liefert die acht niederwertigen Bits des konstanten Ausdruckes, der dem Operator folgt, zurück. Die Größe des Ausdruckes muß dem Datentyp Word entsprechen.

+ »Unäres Plus« – liefert den folgenden Ausdruck unverändert zurück. Der Operator dient lediglich zur expliziten Darstellung positiver Zahlen.

- »Unäres Minus« – liefert den negierten Wert eines Ausdruckes zurück. Der Ausdruck muß ein absoluter, konstanter Wert sein.

: »Segment-Vorgabe« – weist den integrierten Assembler an, die Adresse des folgenden Ausdruckes relativ zu einem bestimmten Segment zu berechnen. Das Segment ist durch den Register-Namen (CS, DS, SS oder ES) vor dem Doppelpunkt bestimmt. Das Ergebnis ist eine Speicher-Referenz und wird aus dem Ausdruck nach dem Doppelpunkt errechnet. Bei einer Segment-Vorgabe erzeugt der integrierte Assembler einen Präfix, um die CPU anzuweisen, den folgenden Ausdruck auf ein anderes als das voreingestellte Segment-Register zu beziehen.

OFFSET liefert den Offset-Anteil des folgenden Ausdruckes zurück. Das Ergebnis ist eine Konstante.

SEG liefert den Segment-Anteil des folgenden Ausdruckes zurück. Das Ergebnis ist eine Konstante.

TYPE Liefert den Typ (Größe in Bytes) des folgenden Ausdruckes zurück. Bei einer Konstanten hat das Ergebnis den Wert 0. Dieser Operator kann mit der *SizeOf*-Funktion von Pascal verglichen werden.

PTR »Typecast-Operator« – weist den integrierten Assembler an, den Datentyp des folgenden Ausdruckes durch einen anderen zu ersetzen. Der neue Datentyp entspricht dem des Ausdruckes vor dem Operator. Das Ergebnis ist eine Speicher-Referenz, deren Wert bestimmt ist durch den Ausdruck nach dem Operator.

***** »Multiplikation« – beide Ausdrücke müssen absolute, konstante Werte sein. Das Ergebnis ist ebenfalls ein absoluter, konstanter Wert.

/ »Ganzzahlige Division« – beide Ausdrücke müssen absolute, konstante Werte sein. Das Ergebnis ist ebenfalls ein absoluter, konstanter Wert.

MOD »Modulo-Operator« – Rest einer ganzzahligen Division. Beide Ausdrücke müssen absolute, konstante Werte sein. Das Ergebnis ist ebenfalls ein absoluter, konstanter Wert.

SHL »Logisches Linksschieben« – beide Ausdrücke müssen absolute, konstante Werte sein. Das Ergebnis ist ebenfalls ein absoluter, konstanter Wert.

SHR »Logisches Rechtsschieben« – beide Ausdrücke müssen absolute, konstante Werte sein. Das Ergebnis ist ebenfalls ein absoluter, konstanter Wert.

+ »Addition« – die Ausdrücke können konstante Werte oder Speicher-Referenzen sein. Allerdings darf nur einer der beiden Ausdrücke ein relozierbarer Wert sein. Das Ergebnis ist in jedem Fall ein relozierbarer Wert, wenn ein Ausdruck relozierbar ist. Sind beide Ausdrücke Speicher-Referenzen, dann ist auch das Ergebnis eine Speicher-Referenz.

- »Subtraktion« – die Klasse des ersten Ausdruckes kann beliebig sein, der zweite Ausdruck muß ein absoluter, konstanter Wert sein. Die Klasse des Ergebnisses entspricht der Klasse des ersten Ausdruckes.

NOT »Bitweise Negation« – der Ausdruck muß ein absoluter, konstanter Ausdruck sein. Das Ergebnis ist ebenfalls ein absoluter, konstanter Ausdruck.

AND »Bitweise Und-Verknüpfung« – beide Ausdrücke müssen absolute, konstante Werte sein. Das Ergebnis ist ebenfalls ein absoluter, konstanter Wert.

OR »bitweise Oder-Verknüpfung« – beide Ausdrücke müssen absolute, konstante Werte sein. Das Ergebnis ist ebenfalls ein absoluter, konstanter Wert.

XOR »Bit-weise Exklusiv-Oder-Verknüpfung« – beide Ausdrücke müssen absolute, konstante Werte sein. Das Ergebnis ist ebenfalls ein absoluter, konstanter Wert.

assembler-Prozeduren und Funktionen

Neben der Möglichkeit, **asm**-Anweisungen (**asm ... end**) innerhalb normaler **begin...end**-Anweisungen zu verwenden, unterstützt der integrierte Assembler auch sog. **assembler**-Prozeduren und -Funktionen. Diese Routinen benötigen keinen **begin...end**-Block, sondern nur einen **asm...end**-Block. Hier ein Beispiel einer **assembler**-Funktion:

```
function LongMul(X,Y: Integer): LongInt; assembler;
asm
    MOV    AX,X
    IMUL   Y
end;
```

Die **assembler**-Anweisung veranlaßt Turbo Pascal, eine Reihe von Optimierungen vorzunehmen:

■ Der Compiler erzeugt keinen Code, der Werte-Parameter in lokale Variablen kopiert. Dies betrifft sämtliche String-Werte-Parameter, sowie alle Werte-Parameter, deren Größe nicht 1, 2 oder 4 Bytes beträgt. Solche Parameter müssen innerhalb einer Prozedur bzw. Funktion behandelt werden, als seien sie **var**-Parameter.

■ Der Compiler reserviert keinen Speicherplatz für die Funktionsergebnis-Variable – eine Referenz auf das *@Result*-Symbol verursacht einen Fehler. Die einzige Ausnahme von dieser Regel betrifft String-Funktionen, denn bei solchen Funktionen wird das *@Result*-Symbol immer dazu verwendet, einen Zeiger aufzunehmen, der durch die aufrufende Routine alloziert wird.

■ Der Compiler erzeugt für Prozeduren und Funktionen, die weder Parameter noch lokale Variablen haben, keinen Stack-Rahmen.

■ Der automatisch generierte Eintritts- und Austritts-Code einer **assembler**-Prozedur bzw. – Funktion sieht folgendermaßen aus (die einzelnen Anweisungen werden nur dann erzeugt, wenn die nebenstehende Bedingung erfüllt ist):

PUSH	BP	{wenn Locals < 0 oder Params < 0}
MOV	BP, SP	{wenn Locals < 0 oder Params < 0}
SUB	SP, Locals	{wenn Locals < 0}
...		
MOV	SP, BP	{wenn Locals < 0}
POP	BP	{wenn Locals < 0 oder Params < 0}
RET	Params	{immer}

Locals gibt die Größe der lokalen Variablen an und *Params* die Größe der Parameter. Haben sowohl *Locals* wie auch *Params* den Wert null, erzeugt der integrierte Assembler keinen Eintritts-Code, und der Austritts-Code besteht lediglich aus dem *RET*-Befehl.

Bei **assembler**-Funktionen erfolgt die Rückgabe des Funktionsergebnisses auf die folgende Weise:

■ Ordinal-Typen (Integer-, Char- und Aufzählungs-Typen) werden in AL (bei 8-Bit-Werten), in AX (bei 16-Bit-Werten) und in DX:AX (bei 32-Bit-Werten) zurückgegeben.

■ Real-Typen werden in DX:BX:AX zurückgegeben.

■ Single-, Double-, Extended- und Comp-Typen (des 80x87-Prozessors) werden auf dem Koprozessor-Stack in ST(0) zurückgegeben.

■ Zeiger-Typen werden in DX:AX zurückgegeben.

■ String-Typen werden zwischengespeichert. Das *@Result*-Symbol liefert einen Zeiger auf den zwischengespeicherten String.

Die **assembler**-Anweisung ist durchaus mit der **external**-Anweisung vergleichbar – beide Anweisungen gehorchen den selben Richtlinien. Mit den nachfolgenden Beispielen sollen einige Unterschiede zwischen **asm**-Anweisungen in normalen Funktionen und **assembler**-Funktionen verdeutlicht werden. Bei beiden Beispielen wird ein String in Großbuchstaben konvertiert, im ersten Beispiel wird dafür eine **asm**-Anweisung eingesetzt.

Beachten Sie, daß der Werte-Parameter *S* in diesem Fall auf eine lokale Variable verweist, denn der Compiler erzeugt automatisch einen Eintritts-Code für die Funktion, der den aktuellen Parameter in den lokalen Speicherbereich kopiert:

```

function UpperCase (S: String): String;
begin
  asm
    CLD
    LEA    SI,S
    LES    DI,@Result
    SEGSS  LODSB
    STOSB
    XOR    AH,AH
    XCHG   AX,CX
    JCXZ   @3
  @1:
    SEGSS  LODSB
    CMP    AL,'a'
    JB     @2
    CMP    AL,'z'
    JA     @2
    SUB    AL,20H
  @2:
    STOSB
    LOOP   @1
  @3:
  end;
end;

```

Das zweite Beispiel stellt die **assembler**-Version von *UpperCase* dar. Hier wird Str nicht in den lokalen Speicherbereich kopiert, und die Funktion muß *S* als **var**-Parameter behandeln:

```

function UpperCase (S: String): String; assembler
asm
  PUSH    DS
  CLD
  LDS     S,S
  LES     DI,@Result
  LODSB
  STOSB
  XOR    AH,AH
  XCHG   AX,CX
  JCXZ   @3
  @1:
    LODSB
    CMP    AL,'a'
    JB     @2
    CMP    AL,'z'
    JA     @2
    SUB    AL,20H
  @2:
    STOSB

```

```

    LOOP    @1
@3:
    POP     DS
end;
```

Einbinden von Assembler-Code

und Turbo-Pascal-Prozeduren und Funktionen, die vollständig in Maschinensprache geschrieben sind, können über den Compiler-Befehl `{$L Dateiname}` in Pascal-Programme oder Units aufgenommen werden, wenn sie mit einem Assembler in eine Object-Datei (Suffix `.OBJ`) übersetzt worden sind. Die Anzahl der `{$L}`-Befehle pro Programm oder Unit ist nicht begrenzt, für jede Object-Datei ist aber ein eigener Befehl erforderlich.

In Maschinensprache geschriebene Prozeduren und Funktionen müssen innerhalb des Pascal-Quelltextes als **external** deklariert werden. Ein Beispiel:

```
function LoCase(Ch: Char): Char; external;
```

Der Assembler-Quelltext muß alle von Pascal aus verfügbaren Routinen in einem Segment mit einem der Namen `CODE`, `CSEG` oder `X_TEXT` deklarieren, wobei das `X` in dem Segment-Namen `X_TEXT` für einen beliebigen Bezeichner steht (zum Beispiel `MODUL1_TEXT`, `OVR_TEXT` oder einfach nur `_TEXT`). Bezeichner müssen darüber hinaus als `PUBLIC` definiert sein, wenn von Pascal aus darauf zugegriffen wird.

Die Prüfmechanismen von Turbo Pascal greifen bei Maschinenprogrammen nur in sehr begrenztem Rahmen – der Programmierer ist hier selbst dafür verantwortlich, daß die Arbeitsweise der Routine mit ihrer Pascal-Deklaration übereinstimmt (Zahl, Typ und Reihenfolge der Parameter, Codierung als *near* oder *far*, Ergebnistyp bei Funktionen).

Ein Maschinenprogramm kann initialisierte Variablen in einem Segment mit dem Namen `CONST` oder `X_DATA` (mit beliebigem Bezeichner `X`) halten. Uninitialisierte Variable müssen in einem Segment `DATA`, `DSEG` oder `X_BSS` (`X` beliebig) abgelegt sein.

Alle Bezeichner des Pascal-Programms oder der Unit stehen dem Maschinenprogramm zur Verfügung, solange der Assembler-Quelltext entsprechende `EXTRN`-Direktiven enthält. (Auch hier liegt es wieder in der Verantwortung des Programmierers, für eine Übereinstimmung der entsprechenden Angaben zu sorgen.)

Turbo Pascal liest mit `{$L}` angegebene `.OBJ`-Dateien und nimmt dabei eine Konvertierung des Intel-Formats in das interne Codeformat des Compilers vor. Diese Konvertierung kann nur dann erfolgreich ausgeführt werden, wenn der Programmierer die folgenden Regeln beachtet:

- Alle Prozeduren und Funktionen müssen in einem einzigen Segment stehen, das den Namen `CODE`, `CSEG` oder `X_CODE` trägt, alle initialisierten Variablen müssen in einem einzigen Segment des Namens `CONST` oder `X_DATA` stehen,

initialisierte Variablen in DATA, DSEG oder X_BSS. Sämtliche anderen Segment- und/oder GROUP-Direktiven werden ignoriert. Das Code-Segment ist immer BYTE-ausgerichtet, das Daten-Segment beginnt immer an einer WORD-Adresse. Die Deklaration von Segmenten als PUBLIC sowie die Deklaration von Klassennamen (CLASS) ist erlaubt und wird von Turbo Pascal ignoriert.

■ Für die Angabe von Variablenwerten im Segment DATA, DSEG bzw. X_BSS dürfen Sie nur Fragezeichen verwenden, z.B.

```
CountDw?
Buffer      DB      128DUP (?)
```

■ Turbo Pascal ignoriert die Werte aller initialisierten Segmente, außer die vom Code-Segment (CODE, CSEG bzw. X_CODE) und vom initialisierten Daten-Segment (CONST bzw. X_DATA).

■ Byte-orientierte Bezugnahmen auf EXTRN-Symbole sind nicht möglich – die Direktiven HIGH und LOW sind deshalb nicht auf Pascal-Bezeichner anwendbar.

Maschinenprogramme mit dem Turbo Assembler

Der *Turbo Assembler* (TASM) verfügt über eine Reihe von Pseudobefehlen, die die Programmierung von Maschinensprache-Routinen für Turbo Pascal-Programme stark vereinfachen. Eine einzige Anweisung – nämlich *.MODEL TPASCAL* – setzt die Übergabe-Konventionen von Turbo Pascal, definiert die entsprechenden Segmentnamen und erzeugt den Start- bzw. Endcode für Routinen.

Mit dem Pseudobefehl *PROC* lassen sich die Parameter einer Maschinenroutine in derselben Reihenfolge wie in einem Pascal-Programm deklarieren; *RETURNS* stellt einen temporären Stringzeiger auf dem Stack zur Verfügung (dessen Platzbedarf automatisch beim RET xx-Befehl berücksichtigt wird).

Ein kurzer Ausblick:

```
.MODEL TPASCAL
.CODE
; -----
; function MyFunc(i,j: Char): stringexternal }
; -----
MyFunc PROC FAR i:Byte, j:Byte RETURNS Result: Word
PUBLIC MyFunc
                                ; <- Startcode wird automatisch eingesetzt
    les di,result              ; Ergebnis-Adresse -> ES:DI
    mov al,i                   ; Parameter i -> AL
    mov bl,j                   ; Parameter j -> BL
    ....
    ret                        ; <- Endcode wird automatisch eingesetzt
```

Weitere Details finden Sie in Kapitel 7 des *Turbo Assembler Benutzerhandbuches*.

Einige Beispiele

Die folgende Unit enthält zwei in (»konventioneller«, d.h. MASM-kompatibler) Maschinsprache geschriebene Routinen: die Funktion *UpperCase*, die Kleinbuchstaben innerhalb eines Strings in Großbuchstaben verwandelt, und die Funktion *StringOf*, die einen mit vorgegebenen Zeichen gefüllten String bestimmter Länge zurückliefert. Fangen wir mit der Pascal-Deklaration an:

```
unit Strings;

interface

function UpperCase(var S: string): String;
function StringOf(Ch: Char; Count: Byte): String;

implementation

{$L STRS}

function UpperCase; external;
function StringOf; external;

end.
```

Das Maschinenprogramm, in dem die Implementation der Funktionen *UpperCase* und *StringOf* enthalten ist, muß in unserem Beispiel in eine Object-Datei des Namens STRS.OBJ übersetzt werden. Falls Sie einen anderen Namen verwenden wollen, müssen Sie den *{\$L}*-Befehl in der Unit *Strings* entsprechend verändern. Beide Routinen sind als *far* deklariert, weil sie im Interface-Teil der Unit erscheinen:

```
CODE    SEGMENT BYTE PUBLIC
        ASSUME  CS:CODE
        PUBLIC  UpperCase,StringOf    ; für Pascal verfügbar
; -----
; function UpperCase(var S: string): string
; -----
UpperRes    EQU      DWORD PTR [BP+10]
UpperStr    EQU      DWORD PTR [BP+6]
UpperCase   PROC FAR
        PUSH    BP                ; BP sichern
        MOV     BP,SP             ; Stackzeiger -> BP
        PUSH    DS                ; DS-Register sichern
        LDS     SI,UpperStr        ; Start überg. String -> DS:SI
        LES     DI,UpperRes        ; Start Ergebnis -> ES:DI
        CLD                      ; Zählung aufwärts
        LODSB                     ; das ist die Stringlänge
```

```

        STOSB                ; -> als 1.Byte Ergebnis
        MOV     CL,AL        ; Stringlänge -> CX (Zähler)
        XOR     CH,CH
        JCXZ    U3          ; Falls 0 Zeichen -> Ende
U1:     LODSB                ; sonst 1 Zeichen laden
        CMP     AL,'a'      ; kleiner 'a' ?
        JB      U2          ; -> ja, keine Veränderung
        CMP     AL,'z'      ; größer 'z' ?
        JA      U11         ; -> ja, keine Veränderung
        SUB     AL,'a'-'A'   ; sonst "Übersetzung"
U2:     STOSB                ; Zeichen als Ergebnis speichern
        LOOP    U1          ; nächster Durchlauf
U3:     POP      DS          ; DS zurück
        POP     BP          ; BP zurück
        RET     4           ; Parameter vom Stack & Ende
UpperCase ENDP

```

Hinweis: `UpperCase` erzeugt keine temporäre Kopie des übergebenen Parameters `S`, der aus diesem Grund als **var** deklariert sein muß.

```

; -----
; function StringOf(Ch: Char; Count: Byte): string
; -----
StrOfRes      EQU     DWORD PTR [BP+10]
StrOfChar     EQU     BYTE PTR [BP+8]
StrOfCount    EQU     BYTE PTR [BP+6]
StringOf      PROC FAR
        PUSH     BP          ; BP sichern
        MOV     BP,SP        ; Stackzeiger -> BP
        LES     DI,StrOfRes   ; Startadresse Ergebnis -> ES:DI
        MOV     AL,StrOfCount ; Anzahl (Parameter)
        CLD                ; Zählung aufwärts
        STOSB                ; Längenbyte ins Ergebnis
        MOV     CL,AL        ; und CX als Zähler
        XOR     CH,CH
        MOV     AL,StrOfChar  ; zu wiederholendes Zeichen (Parm)
        REP     STOSB        ; CX Zeichen ins Ergebnis
        POP     BP          ; BP zurück
        RET     4           ; Parameter vom Stack & Ende
StringOf      ENDP

CODE    ENDS
        END

```

STRS.ASM und die Unit *Strings* werden mit den folgenden Befehlen (auf der DOS-Kommandoebene) übersetzt:

```

TASM STRS      <- übersetzt STRS.ASM und erzeugt STRS.OBJ
TPC STRINGS    <- übersetzt STRINGS.PAS, bindet STRS.OBJ ein und
                erzeugt STRINGS.TPU

```

Das nächste Beispiel zeigt, wie Maschinenprogramme auf Routinen und Variablen von Pascal zugreifen können: Das Programm *Numbers* liest maximal 100 Integerzahlen ein und ruft danach die in Maschinensprache geschriebene Prozedur *CheckRange* auf, die jeden dieser Werte überprüft. Wenn ein Wert außerhalb des vorgegebenen Bereichs liegt, dann ruft *CheckRange* die Pascal-Prozedur *RangeError* zur Ausgabe einer entsprechenden Meldung auf, wobei der entsprechende Wert als Parameter übergeben wird.

```

program Numbers;
{$L CHECK}
var
    Buffer: array[1..100] of Integer;
    Count: Integer;
procedure RangeError(N: Integer);
begin
    WriteLn(N:6, ' liegt außerhalb des Bereichs');
end;

procedure CheckRange(Min,Max: Integer); external;

begin { Hauptprogramm }
    Count := 0;
    while not Eof and (Count<5) do { hier zu Testzwecken nach fünf }
    begin { Werten beendet }
        Inc(Count);
        ReadLn(Buffer[Count]);
    end;
    CheckRange(-10,10); { Überprüfung der Werte }
end.

```

Wir haben hier den Namen CHECK zusammen mit dem {\$L}-Befehl angegeben: der folgende Assembler-Quelltext muß also in eine Datei des Namens CHECK.OBJ übersetzt werden. Im Gegensatz zum vorangegangenen Beispiel ist die Maschinensprache-Prozedur *CheckRange* innerhalb eines Programms deklariert und wird deshalb als **near** aufgerufen.

```

DATA    SEGMENT WORD PUBLIC
        EXTRN    Buffer:WORD,Count:WORD    ; Pascal-Variablen
DATA    ENDS
CODE    SEGMENT BYTE PUBLIC
        ASSUME   CS:CODE,DS:DATA
        EXTRN    RangeError:NEAR    ; Pascal-Routine
        PUBLIC   CheckRange    ; Maschinenroutine

CheckRange    PROC    NEAR
        MOV      BX,SP            ; Zeiger auf die Parameter -> BX
        MOV      AX,SS:[BX+4]    ; Parameter Min -> AX
        MOV      DX,SS:[BX+2]    ; Parameter Max -> DX
        XOR      BX,BX            ; BX auf 0
        MOV      CX,Count        ; Anzahl
        JCXZ     SD4              ; Ende wenn 0

```

```

SD1:  CMP    Buffer[BX],AX    ; zu klein?
      JL     SD2              ; -> ja
      CMP    Buffer[BX],DX    ; zu groß?
      JLE    SD3              ; -> nein
SD2:  PUSH   AX                ; Überschreitung: Register sichern
      PUSH   BX
      PUSH   CX
      PUSH   DX
      PUSH   Buffer[BX]        ; der unzulässige Wert als Parameter
      CALL   RangeError        ; Pascal-Aufruf
      POP    DX                ; Register zurück
      POP    CX
      POP    BX
      POP    AX
SD3:  INC     BX                ; nächstes Element
      INC     BX
      LOOP   SD1                ; nächster Durchlauf
SD4:  RET     4                ; Stack aufräumen & Ende
CheckRange ENDP

CODE   ENDS
      END

```

Ein weiterer Ausblick

Der folgende – alternative – Quelltext für die Prozedur *CheckRange* nutzt die Möglichkeiten des *Turbo Assembler* und demonstriert dabei, wieviele Formalitäten sich mit diesem Programm einsparen lassen:

```

.MODEL TPASCAL                ; Konventionen von Turbo Pascal
LOCALS @@                     ; Präfix für lokale Symbole: @@
.DATA
EXTRN  Buffer:WORD,Count:WORD  ; Pascal-Variablen
.CODE
EXTRN  RangeError:NEAR        ; Pascal-Routine
PUBLIC CheckRange              ; Maschinenroutine

CheckRange PROC NEAR Min:WORD, Max:WORD
      MOV    AX,Min             ; Parameter Min -> AX
      MOV    DX,Max             ; Parameter Max -> DX
      XOR     BX,BX              ; BX auf 0
      MOV     CX,Count           ; Anzahl
      JCXZ    @@4                ; Ende wenn 0
@@1:  CMP     Buffer[BX],AX        ; zu klein?
      JL      @@2                ; -> ja
      CMP     Buffer[BX],DX        ; zu groß?
      JLE     @@3                ; -> nein

```

```

@@2:   PUSH    AX                ; Überschreitung: Register sichern
       PUSH    BX
       PUSH    CX
       PUSH    DX
       PUSH    Buffer[BX]        ; der unzulässige Wert als Parameter
       CALL    RangeError       ; Pascal-Aufruf
       POP     DX                ; Register zurück
       POP     CX
       POP     BX
       POP     AX
@@3:   INC     BX                ; nächstes Element
       INC     BX
       LOOP    @@1              ; nächster Durchlauf
@@4:   RET
CheckRange   ENDP
           END

```

Wie bereits im letzten Abschnitt erläutert, kümmert sich der *Turbo Assembler* nach dem Befehl `.MODEL TPASCAL` automatisch um die Deklaration von Segmenten sowie den Start- und Endcode der Routinen.

Maschinencode mit *inline*

Für sehr kurze Unterprogramme in Maschinsprache sind in Turbo Pascal keine eigenen Object-Dateien notwendig: über **inline** lassen sich einzelne Bytefolgen direkt in den Pascal-Quelltext eines Programms einfügen.

inline-Anweisungen

bestehen aus dem reservierten Wort **inline**, gefolgt von einem Klammerpaar mit einem oder mehreren Elementen, die untereinander durch Schrägstriche voneinander getrennt sind – so, wie es das folgende kurze Beispiel für diesen langen Satz demonstriert:

inline(10/\$2345/Count+1/Data-Offset);

Formal gesehen hat **inline** die Syntax:

Jedes Element besteht aus einer optionalen Größenangabe (< oder >) und

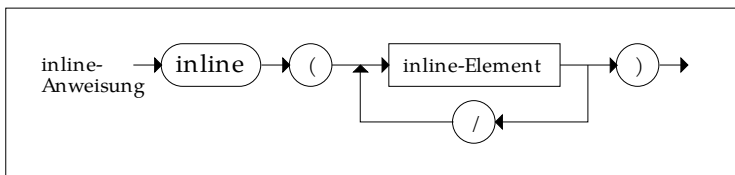


Abbildung 15.1 Syntax von *inline*

einer Konstanten oder einem Variablenbezeichner. Über ein nachfolgendes + oder - zusammen mit einer Konstanten sind relative Angaben möglich:

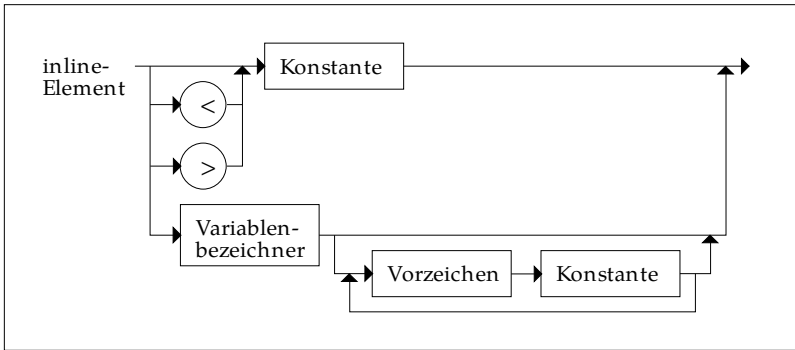


Abbildung 15.2 Syntax eines inline-Elements

Der Wert jedes **inline**-Elements ergibt sich aus den angegebenen Konstanten und/oder den Adressen der angegebenen Bezeichner. Wenn sich die Angabe ausschließlich aus Konstanten zusammensetzt und das Gesamtergebnis im Bereich von 0..255 liegt, dann erzeugt **inline** ein einzelnes Byte. Ergebnisse außerhalb dieses Bereichs oder Angaben von Variablennamen erzeugen grundsätzlich ein Wort (2 Bytes), dessen niederwertiger Teil auf der niedrigeren Adresse gespeichert wird.

Über die Operatoren < und > läßt sich diese automatische Größenbestimmung außer Kraft setzen: **inline**-Elemente, die mit < eingeleitet werden, speichern nur den niederwertigen Teil ihres Ergebnisses mit einem Byte, die Einleitung mit > erzeugt immer ein Wort (dessen höherwertiger Teil unter Umständen den Wert 0 hat). Der Befehl

```
inline(<$1234/>$44);
```

erzeugt also drei Bytes Code, nämlich \$34 (den niederwertigen Teil von \$1234), \$44 und \$00.

Bei der Angabe eines Variablennamens erzeugt **inline** nicht den (zur Compilierungszeit sowieso unbekannten) Wert dieser Variablen, sondern eine Adresse mit 16 Bit, die sich auf das jeweilige Segment bezieht. Globale Variablen (die außerhalb aller Prozeduren und Funktionen bzw. im Interface-Teil einer Unit deklariert sind), werden vom Compiler im Datensegment gespeichert, auf das über das Register DS zugegriffen werden kann. Wenn wir annehmen, daß die Variable *Zaehler* ab der Adresse \$4628 im Daten-Segment steht, dann erzeugt eine Angabe wie

```
inline(Zaehler);
```

den Wert \$4628, der mit zwei aufeinanderfolgenden Bytes (\$28, \$46) gespeichert wird.

Lokale Variablen (die innerhalb einer Prozedur oder Funktion deklariert sind) werden im Stack-Segment gespeichert. Hier berechnet **inline** nicht eine »abso-

lute« Adresse, sondern erzeugt automatisch einen Relativwert zum Inhalt des BP-Registers.

Hinweis: Die Register BP, SP, SS und DS dürfen durch **inline**-Anweisungen nicht verändert werden – alle anderen Register stehen dem Programmierer frei zur Verfügung.

Die im folgenden Beispiel gezeigte Prozedur *FillWord* erwartet drei Parameter: einen Zeiger mit 32 Bit auf den zu füllenden Speicherbereich (*Ziel*), die Größe dieses Bereichs in (16 Bit-) Worten (*Zahl*) und den Wert, mit dem der Bereich gefüllt werden soll (*Datum*):

```
procedure FillWord(var Ziel ^Word; var Zahl, Datum: Word);
begin
  inline (
    $C4/$E/Ziel/    { LES  DI,Ziel[BP] }
    $8B/$E/Zahl/    { MOV  CX,Zahl[BP] }
    $8B/$E/Datum/   { MOV  AX,Datum[BP] }
    $FC/            { CLD   }
    $F3/$AB);       { REP  STOSW }
end;
```

inline-Anweisungen können frei mit anderen Pascal-Befehlen innerhalb des jeweiligen Blocks gemischt werden.

inline-Deklarationen

haben dieselbe Syntax wie **inline**-Anweisungen und im Prinzip denselben Zweck;

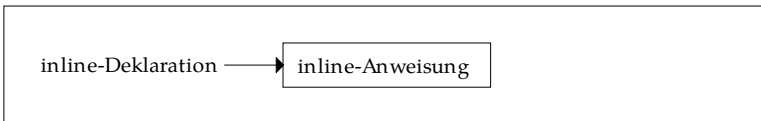


Abbildung 15.3 Syntax einer *inline*-Deklaration

Für normale Prozeduren und Funktionen – unabhängig davon, ob sie **inline**-Anweisungen enthalten oder nicht – erzeugt der Compiler Code, durch den die entsprechenden Parameter auf den Stack gebracht werden, und hängt schließlich einen CALL-Befehl für den eigentlichen Aufruf an. Als **inline** deklarierte Routinen werden dagegen wie Assembler-Makros behandelt; anstatt sie über einen CALL-Befehl aufzurufen, wiederholt der Compiler einfach die mit **inline** angegebenen Befehle jedesmal, wenn der Bezeichner dieser Routine im Quelltext erscheint.

Ein Beispiel:

```
procedure Interrupts_Aus;
inline($FA);    { CLI }

procedure Interrupts_Ein
begin
```

```

inLine($FB); { STI }
end;

```

Interrupts_Ein ist eine völlig normale Prozedur, die mit Befehlen zur Sicherung der Register beginnt, danach den eigentlichen Code ausführt und vor dem Rücksprung die Register wiederherstellt. Der Compiler erzeugt für jedes Vorkommen des Bezeichners *Interrupts_Ein* einen CALL-Befehl. Anders dagegen bei *Interrupts_Aus*: hier fügt der Compiler anstelle eines Aufrufs einfach das Byte \$FA in den laufenden Code ein.

Als **inline** deklarierte Prozeduren und Funktionen arbeiten ohne Start- und Endcode; da sie jeweils direkt in das Programm eingefügt werden, dürfen sie keine Rücksprungbefehle enthalten. Die Übergabe von Parametern ist erlaubt, sie werden vom Compiler wie bei einem normalen Aufruf auf den Stack gelegt und müssen innerhalb der Routine auch wieder entfernt werden. Was dagegen nicht funktioniert, ist eine symbolische Bezugsname über die Parameternamen: ohne Startcode ist keine Adressierung über das BP-Register möglich. Ein weiteres Beispiel:

```

function LongMult(X, Y: Integer): LongInt;
inline(
    $58/      { POP   DX   ; Parameter X }
    $5A/      { POP   AX   ; Parameter Y }
    $F7/$EA); { IMUL  DX   ; X * Y -> DX:AX }

```

Für jeden Aufruf der Form

```
Z := LongMult(A,B);
```

erzeugt der Compiler Befehle, mit denen die Werte von *A* und *B* auf den Stack gelegt werden, fügt dann direkt die vier Bytes \$58/\$5A/\$F7/\$EA in den laufenden Code ein und hängt schließlich Befehle an, die den Inhalt des Registerpaars DX:AX in der Variablen *Z* speichern. Weitere Befehle (spezifisch: Start- und Endcode einer Prozedur oder Funktion) werden nicht erzeugt.

Deklarationen mit **inline** sind ausschließlich für sehr kurze Routinen (ca. 10 Bytes) gedacht. Da der Compiler ihren Code jedesmal direkt einfügt, haben sie im Gegensatz zu anderen Bezeichnern keine feste Adresse und können nicht als Argumente der Funktionen *Addr*, *Ofs* und *Seg* oder als Operand von @ verwendet werden.

Index

@Code 246
 @Data 246
 @Operator 70
 @Result 246
 {\$A} (Compiler-Schalter) 231
 Back (Turtlegrafik-Prozedur) 173
 Forwd (Turtlegrafik-Prozedur) 173
 \$L 208
 \$R 203
 8087 (Coprozessor)
 Emulator 175
 Prüfung 181
 _BSS (Segmentname) 258
 _DATA (Segmentname) 258
 _TEXT (Segmentname) 257
 {\$E} (Compiler-Schalter) 175
 {\$F+} (Compiler-Schalter) 105
 {\$N} (Compiler-Schalter) 175

A

absolute Variablen 45
 Adreß-Operator 70
 Adressübergabe 216
 Aktivierung
 mit vollständigem Bezeichner 79
 Anweisung
 Prozedur 79
 with 49
 Anweisungen 77
 bedingte 82
 case 83
 einfache 77
 goto 80
 if 82
 Prozeduraufrufe 79
 strukturierte 81
 Verbund- 81
 Wiederholung 85
 with 88

Zuweisungen 78
 Arrays
 als Parameter 217
 Speicherung 198
 Typen 27
 typisierte Konstanten 55
 und Prozedur-Variablen 106
 ASCIIZ-Format 200
 Assembler-Code
 mit dem integrierten Assembler 235
 und Turbo Pascal 257
 assembler-Deklaration 95
 Attribute von Dateien 138
 Aufruf
 Konventionen 79
 mit vollständigem Bezeichner 79
 virtueller Methoden 204
 Aufrufe
 near und far 218
 von Funktionen 72
 von Funktionen und Prozeduren 215
 von Prozeduren 79
 von Unterprogrammen 102 f.
 Aufzählungstypen 22
 als Parameter 216
 Speicherung 195
 Ausdrücke 61
 absolute 250
 boolesche 233
 Konstanten-Vorausberechnung 228
 Mengen-Konstruktoren 73
 Operatoren 61, 65 ff.
 Reihenfolge der Berechnung 229
 relozierbare 250
 Syntax 61
 Umwandlung von Wert-Typen 74
 und Prozedur-Variablen 75
 Ausgabe
 Spezialzeichen 142

- Text im Grafikmodus 151
- Ausschluß
 - unbenutzter Daten 231
- Ausschnitte von Grafikbildern 152
- Austritts-Code 255
- Auswertung von Ausdrücken
 - Compiler-Schalter 67, 229
- B**
- bedingte Anweisungen 82
- Befehle 77
- Belegung, dynamische 187
- Berechnung
 - Reihenfolge 229
- Bereich 30
- Bezeichner 5
 - einer Methode 49
 - Gültigkeitsbereich (Scope) 15 f.
 - Methode 35
 - qualifizierte 6
 - Standard- 5
 - vollständige mit Self 80
- Bildschirm
 - Farben 152
 - Prozeduren und Funktionen 147
 - Seiten in der Grafik 152
- Bindung
 - dynamische 33
- Blöcke 13
 - Gültigkeitsbereich (Scope) 15
 - Syntax 13
- Boolean (Datentyp) 22
 - als Parameter 216
 - Speicherung 195
- boolesche Operatoren 66
- Byte (Datentyp)
 - als Parameter 216
 - Speicherung 194
- C**
- case-Anweisung 83
- CBreak (Variable) 172
- CGA-Karte
 - hochauflösender Modus 154
 - Speicherorganisation 153
- Char (Datentyp) 22
 - als Parameter 216
 - Speicherung 194
- CheckBreak (Variable) 144
- CheckEof (Variable) 145
- CheckSnow (Variable) 145
- ClearScreen (Turtlegrafik-Prozedur) 173

- Clipping 152
- Close-Funktion bei Gerätetreibern 227
- CODE (Segmentname) 257
- COM (Geräteeinheiten) 125
- Comp (Datenformat) 177
- Comp (Datentyp)
 - als Parameter 217
 - Speicherung 197
- Compiler
 - automatische Optimierung 227
 - Schalter 25, 44, 67, 229, 233
- Compiler-Anweisung
 - \$L 208
 - \$R 203
- CON (Geräteeinheit) 124
- CONST (Segmentname) 258
- constructor (Schlüsselwort) 98
- Coprozessor
 - 8087 (Coprozessor) 175
 - Ausgabe von Realzahlen 180
 - Datentypen 176
 - Prüfung 181
 - Rechenstack 179
 - und Units 180
 - Variable Test8087 182
- Crt (Standard-Unit) 141
- CSEG (Segmentname) 257
- Cursor
 - im Grafikmodus 150

- D**
- DATA (Segmentname) 258
- Datei-Typen 36
 - Speicherung 199
- Dateien
 - Attribute 138
 - Close-Funktion 227
 - Eröffnen 119
 - Flush-Funktion 226
 - InOut-Funktion 226
 - Modus-Konstanten 137
 - Öffnungsmodus 123
 - Open-Funktion 226
 - Schließen 120
 - Typen und Records 137
- Datensegment 44, 185
- Datentypen des Coprozessors 176
- DateTime (Record-Typ) 138
- Datum und Uhrzeit
 - Prozeduren 140
- Dec (Prozedur)
 - optimiert 233

- Deklaration
 - Methode 33, 98
 - Objekttypen 30
 - von Funktionen 96
 - von Prozedurtypen 38
 - von Variablen 43
- Deklarationen
 - von Prozeduren 91
- destructor (Schlüsselwort) 98
 - Aufrufkonventionen 206
 - Definition 99
- DirectVideo (Variable) 145
- Disketten- und Festplattenlaufwerke
 - Statusfunktionen 140
- Dispose 128
- Dispose (erweiterte Syntax) 34
- Dispose (Prozedur)
 - Arbeitsweise 189
- Dispose (Standardprozedur)
 - Destruktor als Parameter 206 f.
 - erweiterte Syntax mit Destruktoren 99
 - VMT 203
- DOS
 - Geräteeinheiten 124
 - Parameter ermitteln/setzen 141
 - und Hardware-Interrupts 224
- Dos (Standard-Unit) 136
- DosError (Variable) 139
- Double (Datenformat) 176
- Double (Datentyp)
 - als Parameter 217
 - Speicherung 196
- DS (Prozessor-Register) 185
- DSEG (Segmentname) 258
- dynamische Bindung 33
- dynamische Objektinstanzen 206
- dynamische Speicherverwaltung 187
 - durch Graph 158
 - Fehlerbehandlung 193
- E**
- EGA-Karte 154
- Einfache
 - Typen 20
 - typisierte Konstanten 54
- Eingabe
 - Steuerzeichen 143
 - zeilenweise 143
- Eintritts-Code 255
- Emulation des Coprozessors 175
 - in Maschinenprogrammen 181
- Environment ('Umgebung')
 - Variable '87' 181
- Environment-Tabelle
 - Funktionen 141
- ErrorAddr (Variable) 221
- Ersatzmethode 33
- Exit-Prozeduren 220
- ExitCode (Variable) 221
- ExitProc (Variable) 220
- Extended (Datenformat) 177
 - Operationen mit 177
- Extended (Datentyp)
 - als Parameter 217
 - Ausgaben 180
 - Speicherung 197
- external (Schlüsselwort) 208
- external-Deklaration 95
- F**
- Fail (Standardprozedur) 212
- far
 - Aufruf 218
 - Deklaration 92
 - Prozedur-Variablen 105
- Farben 152
 - für Text und Hintergrund 144
- Farb-Paletten 152 ff.
- Fehler
 - bei numerischen Vergleichen 178
- Fehlerbehandlung
 - Aufruf virtueller Methoden 203
 - Fail 212
 - für den Heap 193
 - heaperror 211
 - in der Unit Graph 155
 - in Konstruktoren 211
 - Variablen ErrorAddr und ExitCode 221
- Feld
 - Gültigkeitsbereich 32
 - Objekt 30
- Fenster 142
 - Koordinaten 146
 - und Grafik 152
- FileMode (Variable) 123
- FileRec (Datentyp) 199
- FileRec (Record-Typ) 137
- Flächenfüllung 151
- Flag-Konstanten 136
- Fließkomma
 - Dezimalpunkt 195
 - Operationen(80(2)87) 25
- Fließkomma-Konstanten
 - Speicherung 177

- Fließkomma-Operationen
 - Software 25
- Fließkommawerte 175 f.
- Flush-Funktion bei Gerätetreibern 226
- for-Anweisung 86
- Formate bei Integern 21
- forward-Deklaration 94
- Fragmentierung des Heaps 189
- Fragmentliste der dynamischen Speicherverwaltung 191
- FreeMem (Prozedur)
 - Arbeitsweise 189
- Freigabe dynamischer Variablen 188
- Funktion
 - Sizeof 254
- Funktionen
 - Aufruf 72
 - Aufruf und Parameterübergabe 215
 - Deklaration 96
 - Deklaration als Inline 266
 - eingeschachtelte 218
 - Ergebnisse 217
 - Start- und Endcode 219
- Funktionsergebnis
 - Rückgabe 255

G

- Geräte 124
- Gerätetreiber
 - Close-Funktion 227
 - Flush-Funktion 226
 - für Textdateien 224
 - InOut-Funktion 226
 - Open-Funktion 226
- GetMem (Prozedur)
 - Arbeitsweise 187
 - Fehlerbehandlung 193
- globale Variablen 44
 - Speicherung 265
- goto-Anweisung 80
- Grafik
 - Beispielprogramme 156
 - Cursor 150
 - Fehlerbehandlung und -codes 155
 - Fenster und Ausschnitte 152
 - Kompatibilität mit Version 3.0 172
 - Koordinatensystem 150
 - mehrere Bildschirmseiten 152
 - Objekte und Zeichenstil 151
 - Prozeduren und Funktionen 165
- Grafiktreiber 148
- Graph (Standard-Unit) 148

- Graph3 (Standard-Unit) 172
- GraphFreeMem 158
- GraphGetMem 158
- GraphResult (Prozedur) 155
- Gültigkeitsbereich 98
 - Methode 98

H

- Heading (Turtlegrafik-Funktion) 173
- Heap-Bereich 187
 - Belegung durch Graph 158
 - Fragmentierung 189
 - Fragmentliste 191
 - Spitze 187
 - Standardvorgaben 187
- Heaperror 211
- HeapError (Variable) 193
- HeapPtr (Variable) 187
- HideTurtle (Turtlegrafik-Prozedur) 173
- HighVideo (Prozedur)
 - in Turbo3 172
- Hintergrundfarbe 144, 153
 - für CGA-Karten 154
 - Konstanten in graphics.h 153
- Home (Turtlegrafik-Prozedur) 173

I

- I/O-Adressen
 - direkter Zugriff 227
- IBM8514-Adapter 155
- identische Typen 39
- if-Anweisung 82
- Implementierung
 - Methode 33
- Inc (Prozedur)
 - optimiert 233
- Indizierungen 47
 - Index-Prüfung 230
 - Optimierung 230
- Initialisierung
 - der Farb-Palette 155
 - des Stack-Registers 185
- inline
 - Anweisungen 264
 - Deklaration 96, 266
 - Prozedur-Variablen 105
- InOut-Funktion bei Gerätetreibern 226
- Input (Dateivariablen) 135
- Input (Standard-Datei)
 - und Crt 141
- Instanz
 - Objekt 34

- Integer (Datentyp) 21
 - als Parameter 216
 - Speicherung 194
- interrupt
 - Deklaration 93
 - und Prozedur-Variablen 105
- Interrupts
 - Bearbeitung 222
 - Hardware 224
 - Prozeduren 140, 222
 - und DOS 224
 - Vektoren des Systems 135
- IOResult (Funktion)
 - Statusvariable 134
- K**
- Kbd (Variable) 171
- Kommentare 11, 236
- Kompatibilität 35, 39, 78
- Konstanten 10, 53
 - berechnete 10
 - einfach typisiert 54
 - Fließkomma 177
 - gemeinsame Speicherung 228
 - in der Unit Crt 143
 - in der Unit Dos 136
 - in der Unit Graph 160
 - in der Unit Turbo3 170
 - Optimierung bei Multiplikationen 230
 - typisierte 185
 - von Objekten 57
 - Vorausberechnung 228
- Konstruktor 98
 - Aufrufkonventionen 206
 - Definition 99
 - Fail (Fehlerbehandlung) 211 f.
 - Fehlerbehandlung 211
 - und VMT 201, 204
 - vererbter 33
- Konvertierung von Formaten 21
- Koordinatensystem von Graph 150
- Kurzschlußverfahren 67, 229
- L**
- Labels 7, 237
 - lokale 237
 - normale 238
- LastMode (Variable) 145
- Laufzeitbibliothek
 - Code-Segment 185
- Linker
 - unbenutzte Programmteile 231
- logische Operatoren 66
- lokale Variablen 44
 - Speicherung 265
- LongFilePos, Long File Size (Funktionen)
 - in Turbo3 172
- LongInt (Datentyp)
 - als Parameter 216
 - Speicherung 194
- LongSeek, Low Video (Prozedur)
 - in Turbo3 172
- LPT (Geräteeinheit) 125
- M**
- Mark (Prozedur)
 - Arbeitsweise 188
- Maschinencode
 - mit Inline 264
- Maschinenprogramme
 - und der Emulator 181
 - und Turbo Assembler 258
- Maschinensprache
 - mit dem integrierten Assembler 235
 - mathematischer Coprozessor 175
- MaxAvail (Funktion)
 - in Turbo3 172
- Mem, MemW, MemL (Pseudo-Arrays)
 - 201
- MemAvail (Funktion)
 - in Turbo3 172
- Mengen
 - als Parameter 217
 - Ersatz durch boolesche Ausdrücke 233
 - Speicherung 198
 - typisierte Konstanten 57
- Mengen-Typen
 - Konstruktor 73
 - Operatoren 68
 - Vergleiche 70
- Methode
 - Aufruf als Funktion oder Prozedur 79
 - Aufruf einer statischen 35
 - Aufruf einer virtuellen 35, 204
 - Aufrufkonventionen 205
 - Bezeichner 35, 49, 79
 - Definition 30
 - Deklaration 33, 98
 - Ersatzmethode 33
 - externe 208
 - Gültigkeitsbereich 98
 - Implementierung 33, 98
 - in Assembler 208
 - Kompatibilität von Parametern 102

- Parameter 79
 - statisch 33
 - virtuell 33, 79
 - vollständiger Bezeichner 33, 49, 79

- MSB 195

- Multiplikations-Optimierung 230

N

- Nachkomme 30

- NaN-Werte 196

- near

- Deklaration 92

- Aufruf 218

- New 37, 128

- New (erweiterte Syntax) 34

- New (Prozedur)

- Arbeitsweise 187

- Fehlerbehandlung 193

- erweiterte Syntax 206

- erweiterte Syntax mit Konstruktoren 99

- Konstruktor als Parameter 206

- VMT 203

- nil (Zeiger-Konstante)

- Wert 198

- Normalisierung von Zeigerwerten 187

- NormVideo (Prozedur)

- in Turbo3 172

- NoWrap (Turtlegrafik-Prozedur) 173

- NUL (Geräteeinheit) 125

- Nullstring 26

- numerische Werte 7

- und der 8087 180

O

- Objekt

- Bereich 30

- Definition eines Destruktors 99

- Definition eines Konstruktors 99

- dynamische Freigabe 99

- dynamische Instanz 206

- dynamische Reservierung 99

- Feld 30

- Feldbezeichnungen 49

- Gültigkeitsbereich 32

- Instanz 34

- internes Datenformat 201

- Konstruktoren und VMT 33, 99, 201 ff.

- Nachkomme 30

- polymorph 35, 78, 102

- Typdeklaration 30

- typisierte Konstanten 57

- Vererbung 30

- VMT 202, 204

- Zeiger auf 35

- Objektformat 201

- Open-Funktion bei Gerätetreibern 26

- Operatoren 65

- boolesche 66

- logische 66

- Rangfolge 61, 252

- relationale 69

- Syntax 61

- unäre 61

- Optimierung

- automatische 227

- Compiler-Schalter 233

- ordinale Typen 20

- Output (Dateivariablen) 135

- Output (Standard-Datei)

- und Crt 141

- Overlay (Standard-Unit) 168

- Overlay-Puffer

- Lage im Speicher 186

- OvrHeapEnd (Variable) 186

- OvrHeapOrg (Variable) 186

- OvrResult (Variable) 168

P

- Pack 128

- Parameter 101, 103

- Adressübergabe 216

- prozedurale 107

- Self 79 f., 98, 205

- Typkompatibilität 102

- untypisierte 103

- Variablen- 102

- VMT 206

- Wert- 102

- Wertübergabe 216

- PenDown (Turtlegrafik-Prozedur) 174

- PenUp (Turtlegrafik-Prozedur) 174

- Peripheriegeräte 124

- Pointer (Datentyp) 37

- Polymorphie

- Typkompatibilität von Parametern 102

- Zeigerzuweisung 35

- Zuweisung einer Objektinstanz 78

- Port, PortW (Pseudo-Arrays) 227

- Präfix-OpCodes 238

- Printer (Standard-Unit) 136

- PRN (Geräteeinheit) 125

- Programmcode

- automatische Optimierung 227

- toter 231

- Programme
 - Gesamtumfang 185
 - manuelle Optimierung 233
 - Programmzeilenlänge 12
 - Prozedur 79
 - Prozedur-Aufruf
 - effizienter 252
 - Prozedur-Variablen 104
 - als Parameter 107
 - in Ausdrücken 75
 - Speicherplatzbedarf 104
 - Typ-Umwandlung 51
 - prozedurale Typen
 - Speicherung 200
 - typisierte Konstanten 58
 - Prozeduren
 - Aufruf und Parameterübergabe 215
 - Aufrufe 79
 - Deklaration als Inline 266
 - eingeschachtelte 218
 - Interrupts 222
 - Maschinensprache 257
 - Start- und Endcode 219
 - zum Beenden eines Programms 220
 - Prozedurtypen 37, 104
 - Deklaration 38
 - Prozesse
 - Prozeduren und Funktionen 140
 - Prozessor-Register
 - verfügbare 220
 - für Funktionsergebnisse 217
 - Prüfung des Stacks 44
 - PUBLIC-Bezeichner 257
- Q**
- Qualifizierer 46
 - qualifizierte Bezeichner 6
- R**
- RandSeed (Variable) 134
 - Real (Datentyp)
 - als Parameter 216
 - Speicherung 195
 - und Unendlichkeiten 196
 - Real-Typen 24
 - Speicherung 195
 - Realzahlen 175
 - 8087-Berechnung 25
 - Ausgabe mit dem 8087 180
 - Extended 177
 - Software-Berechnung 25
 - Vergleiche 178
 - Records
 - als Parameter 217
 - für Dateien 137
 - Speicherung 199
 - typisierte Konstanten 56
 - und Feldbezeichner 48
 - und Prozedur-Variablen 106
 - Register
 - Flags 136
 - Registers (Record-Typ) 138
 - Reihenfolge der Berechnung 229
 - relationale Operatoren 69
 - Release (Prozedur)
 - Arbeitsweise 188
 - Relokation 250
 - repeat-Anweisung 85
 - reservierte Wörter 4
 - Reset (Prozedur)
 - Öffnungsmodi für Dateien 123
 - RET-Befehl 239
 - Rewrite (Prozedur)
 - Öffnungsmodus für Dateien 123
- S**
- Schiebebefehle 230
 - Schleifen
 - for 86
 - repeat..until 85
 - while 86
 - Schlüsselwort
 - constructor 98
 - destructor 98
 - external 208
 - virtual 33
 - with 49
 - SearchRec (Record-Typ) 139
 - SEG-Operator 246
 - Segmentnamen
 - Code-Segment 257
 - Daten-Segmente 258
 - Self 80, 98
 - in Methoden 79
 - Parameter 205 f.
 - SetBkColor (Prozedur) 154
 - SetHeading (Turtlegrafik-Prozedur) 174
 - SetPencolor (Turtlegrafik-Prozedur) 174
 - SetPosition (Turtlegrafik-Prozedur) 174
 - ShortInt (Datentyp)
 - als Parameter 216
 - Speicherung 194
 - ShowTurtle (Turtlegrafik-Prozedur) 174
 - Single (Datenformat) 176

- Single (Datentyp)
 - als Parameter 216
 - Speicherung 196
- SizeOf 204
- Speicher
 - Aufbau 185
 - direkter Zugriff 200 f.
- Speicher-Ausdruck
 - untypisiert 251
- Speicherung
 - globale Variablen 185
 - typisierte Konstanten 185
- Speicherverwaltung, dynamische 187
 - Fragmentliste 191
- Spezialzeichen
 - Ausgabe über Crt 142
- Sprachelemente 3
- Sprunganpassung
 - automatische 239
- Stack
 - Überlauf 185
- Stack-Rahmen 255
- Stack-Register
 - Initialisierung 185
- Stack-Überlauf
 - automatische Prüfung 44
- Stacksegment 44
- Standard-Units 131
 - Crt 141
 - Dos 136
 - Graph 148
 - Graph3 172
 - Overlay 168
 - Printer 136
 - System 133
 - Turbo3 170
- Standardbezeichner 5
- Standardfunktion
 - Sizeof 204
 - TypeOf 204
- Standardprozeduren und -funktionen (Zusammenfassung) 127
- Standardvorgaben
 - für den Heap 187
 - für den Stack 186
- Start- und Endcode 219
- statische Methode 33
- Str (Prozedur)
 - für das Format Extended 180
- String (Datentyp)
 - als Parameter 217
- String-Konstanten 244

- String-Typen 9, 25
 - Operatoren 68
 - Speicherung 198
 - Vergleiche 69
- Strings
 - typisierte Konstanten 54
- strukturierte Typen 26
 - typisierte Konstanten 55
- Syntaxdiagramm 5
- System (Standard-Unit) 133
- System (Unit)
 - und Prozedur-Variablen 105
 - Variable Test8087 182

T

- TASM 258
- Teilbereichstyp 23
- Test8087 (Systemvariable) 182
- Text-Geräteeinheiten 125
- TextAttr (Variable) 146
- Textausgabe
 - im Grafikmodus 151
- Textdatei-Gerätetreiber 224
- Textfarben 144
- TextRec (Datentyp) 199
- TextRec (Record-Typ) 137
- Tokens 3
- Treiber
 - in der Unit Graph 148
- Turbo Assembler 258
- Turbo Pascal
 - interne Details 185
 - und Maschinenprogramme 235, 257
 - Version 3.0 170
- Turbo3 (Standard-Unit) 170
 - Prozeduren und Funktionen 172
- TurnLeft (Turtlegrafik-Prozedur) 174
- TurnRight (Turtlegrafik-Prozedur) 174
- TurtleDelay (Turtlegrafik-Prozedur) 174
- Turtlegrafik
 - Unit Graph3 172
- TurtleThere (Turtlegrafik-Funktion) 174
- TurtleWindow (Turtlegrafik-Prozedur) 174
- Typ-Bezeichner 249
- Typ-Prüfung 251
- Typ-Umwandlung
 - von Variablen 49 ff.
 - von Werten 74
- Typdeklaration
 - Objekt 30
- Typen 19

Definition 41
 einfache 20
 identische und kompatible 38
 in der Unit Crt 143
 in der Unit Dos 136
 in der Unit Graph 160
 in der Unit Turbo3 170
 kompatible 39
 Objekt 30
 ordinale 20
 prozedurale 37
 Real 24
 Strings 25
 strukturierte 26
 TypeOf 204
 typisierte Konstanten 53

U

Übergabe
 Adressen 216
 von Funktionsergebnissen 217
 von Parametern 215
 Werte 216
 Überlauf des Stacks 185
 unäre Operatoren 61
 Unendlichkeiten 196
 Units
 Abhängigkeiten 115
 Bezeichner 249
 gemeinsame Datendeklarationen 117
 Implementationsteil 115
 Reihenfolge der Speicherung 185
 residente 110
 Standard 131
 und der 8087 180
 Unterbereiche 23
 untypisierte Zeiger 37
 Ursprung 150

V

var-Parameter 216, 248
 Variablen 43
 absolute 45
 Bezug 46
 CBreak 172
 Deklaration 43
 DosError 139
 dynamische 49, 188
 ErrorAddr 221
 ExitCode 221
 ExitProc 220
 Funktionsergebnis 255

globale 44, 185, 265
 in der Unit Crt 143
 in der Unit Dos 136
 in der Unit Graph 160
 in der Unit System 133
 in der Unit Turbo3 170
 Indizierungen 47
 Kbd 171
 lokale 44, 247, 265
 Parameter 102, 216
 Pointer 49
 Prozedur- 104
 Records und Feldbezeichner 48
 Typ-Umwandlung 49
 Typen 37
 Vektor-Zeichensätze 151
 Verbundanweisung 81
 Vererbung 30
 Vergleichsoperatoren 69
 Verzeichnisse
 Prozeduren 140
 VGA-Karte 154
 Video-Adapter
 mehrere 149
 Videomodus
 Konstanten 143
 virtual (Schlüsselwort) 33
 virtuelle Methode 33
 Aufruf 35, 204
 VMT 202
 Aufbau 203
 interne Verarbeitung 201
 Zeigerinitialisierung 204
 Parameter 206
 Vorausberechnung von Konstanten 228

W

Wertebereiche
 Prüfung 230
 Wertparameter 102
 Wertübergabe 216
 while-Anweisung 86
 WindMax, WindMin (Variablen) 146
 with (Schlüsselwort) 49
 with-Anweisung 88
 Word (Datentyp)
 als Parameter 216
 Speicherung 194
 wortweise Ausrichtung 231
 Wrap (Turtlegrafik-Prozedur) 174
 Write (Prozedur)
 für das Format Extended 180

Writeln (Prozedur)
für das Format Extended 180

X

XCor (Turtlegrafik-Funktion) 174

Y

YCor (Turtlegrafik-Funktion) 174

Z

Zeichenausgabe
über Crt 142
Vektor-Zeichensätze 151
Zeichenfarbe 153
Zeichenstil (Grafik) 151

Zeiger

@Result 255
als Parameter 217
normalisierte Form 70
Normalisierung 187
typisierte Konstanten 58
untypisierte 37
Zuweisungskompatibilität 35
Zeiger-Typen
Speicherung 198
Vergleiche 70
Zuweisung 78
Kompatibilität 40, 78
von Prozedur-Variablen 75

Gestärkt durch die Leistungsfähigkeit und Effektivität der objektorientierten Programmierung gleibt die Stellung von Turbo Pascal als Marktführer auch in Zukunft gesichert... Sicher ist nur eins, OOP wird neue Bereiche eröffnen, sei es in Datenbank-Management-Systemen oder auch in Programmiersprachen der Zukunft.

*Jürgen Mertes/Klaus Schäfer
PC Magazin, 18. Oktober 1989*

Das objektorientierte Modell hat das Zeug dazu, nach der Einführung der strukturierten Programmierung vor gut zwei Jahrzehnten die zweite »sanfte« Wende auf dem Gebiet der Software-Entwicklung zu sein.

*Ronald Fischer
Toolbox, Oktober 1989*